HOCHSCHULE RHEIN-WAAL
&
FLUXANA GMBH & CO. KG

BACHELOR'S THESIS

Development of an automated powder dosing system using a 6-DOF collaborative robotic arm (cobot)

by investigating the influence of vibration, angle of dosing, and rotational speed to the mass flow of the powder.

Author:
Abdelrahman MOSTAFA
Matriculation No. 29528

Supervisors:
Prof. Dr. Ronny HARTANTO
Dr. Rainer SCHRAMM

A thesis submitted in fulfillment of the requirements for the Bachelor degree of Science

in the

Mechatronic Systems Engineering
Faculty of Technology & Bionics

December 18, 2023

Declaration of Authorship

I, Abdelrahman MOSTAFA, declare that this thesis titled, "Development of an automated powder dosing system using a 6-DOF collaborative robotic arm (cobot)" and the work presented in it are my own. The submission at hand has been acknowledged appropriately and its sources are listed in the reference section. I confirm that it has not been subjected to scrutiny in either a similar or identical form. Furthermore, it has not undergone publication beforehand.

Signed: _____

Date: _____

HOCHSCHULE RHEIN-WAAL

Abstract

Faculty of Technology & Bionics
Research & Development at Fluxana GmbH & Co. KG

Bachelor of Science

Development of an automated powder dosing system using a 6-DOF collaborative robotic arm (cobot)

by Abdelrahman MOSTAFA

This thesis presents a comprehensive investigation into the flow characteristics of sugar powder within a fully automated system employing a collaborative robotic arm. Conducting over 80 trials, with 48 dedicated to Experiment 1 and 32 to Experiment 2, the study aimed to understand the behavior of sugar powder and test the viability of achieving accurate powder dosage control through vibrations. Results indicate that a 6-DoF cobot can effectively manage a fully automated powder dosing system, with the best average precision recorded at 0.9mg. The motion planning algorithm showcased robustness, successfully avoiding collisions and demonstrating reliability across all trials. The study analyzes the impact of parameters such as frequency, angle, rotation, and mass left on powder mass flow, providing valuable insights. The proposed research addresses the dosing challenges within the industrial sector, particularly focusing on applications in X-ray fluorescence analysis. The collaboration between our developed system described in this paper and BORAMAT® systems will improve the understanding of dosing dynamics, leading to the development of a precise control algorithm for dosing applications of one gram or more.

Acknowledgements

Initially, all praise be to Allah, the Most Compassionate and Most Merciful. Without His blessings, none of this would have been possible.

I want to express my sincere gratitude to Dr. Rainer SCHRAMM, CEO of Fluxana GmbH, for his guidance as my company supervisor during my thesis. His extensive industrial experience and insightful feedback were invaluable in guiding me through the experimental process and analyzing the recorded data.

I would like to extend my gratitude to Professor Prof. Dr. Ronny HARTANTO, who is a Computer Engineering professor at HSRW, for his insightful assistance during the programming of the system. His supervision and guidance on the motion planning algorithm was extremely valuable.

Additionally, I can not forget Eng. Carine Silva Allen, as she was supporting me throughout the research on my daily activities. Your help and support are highly appreciated.

I am immensely grateful to my family, particularly my father and mother, for their unwavering psychological, moral, and financial support from the very beginning of my life until this moment.

Contents

Declaration of Authorship iii

Abstract v

Acknowledgements vii

1 Introduction 1
1.1 Motivation . . . 1
1.2 Objectives . . . 2
1.3 Thesis Structure . . . 2

I Basics of Robotics, ROS, and Powder Dosage 3

2 Basics of Robotics 5
2.1 Types of Robots . . . 6
2.2 Robot Components . . . 8
2.3 Spatial descriptions and transformations . . . 11
2.3.1 Position, Orientation, and Frames . . . 11
Position . . . 11
Orientation . . . 12
Frame . . . 12
2.3.2 TransFormers . . . 12
2.3.3 Jacobian Matrix . . . 12
2.4 Forward & Inverse Kinematics . . . 14
2.4.1 Forward Kinematics . . . 14
2.4.2 Inverse Kinematics . . . 15
2.5 Motion Planning . . . 15
2.5.1 Motion Planning Framework (MPF) . . . 16
2.5.2 URDF . . . 17
URDF Components in Hydro . . . 18
2.6 Conclusion . . . 18

3 Robot Operating System | ROS 19
3.1 Linux for Robotics . . . 19
3.1.1 What Is Ubuntu? and Why for Robotics? . . . 19
3.1.2 Ubuntu File Structure . . . 20
3.2 Philosophy Behind ROS . . . 21
3.3 Preliminaries . . . 22
3.3.1 ROS-Graph [11] . . . 22
3.3.2 Roscore [76] . . . 24
3.3.3 catkin, Workspaces, and ROS Packages . . . 25
3.4 Nodes in ROS . . . 27

3.4.1 Benefits of Using Nodes . . . 27
3.4.2 Node Identification . . . 27
3.4.3 Node Type . . . 27
3.5 ROS Master Communication . . . 28
3.5.1 .bashrc File . . . 29
3.6 Publishers-Subscribers . . . 29
3.6.1 Topics in ROS . . . 30
Key Characteristics: . . . 30
3.6.2 Publishing a topic in ROS node . . . 31
3.6.3 Messages in ROS . . . 32
3.6.4 Subscribing to a topic in ROS node . . . 33
3.7 Services . . . 34
3.7.1 Service-Clients node in ROS . . . 34
3.7.2 *.srv Files . . . 35
3.7.3 Configuration for Custom Service Compilation . . . 35
Modification of CMakeLists.txt . . . 35
Modification of package.xml . . . 37
3.7.4 Service-Server node in ROS . . . 38
3.8 Actions . . . 41
3.9 ROS commands summary . . . 42
3.10 MoveIt! [65] . . . 43

4 Basics of Powder Dosing **45**
4.1 Dependant & Independent Variables . . . 46
4.1.1 Research Hypothesis . . . 47

II Methodology, Experimental Set-up, and Results **49**

5 Methodology **51**
5.1 Sequence Logic . . . 51
5.2 Important Fuctions from FX_ROS Package . . . 52
5.3 State Diagram . . . 53
5.3.1 Zero State . . . 53
5.3.2 Calibration . . . 54
5.3.3 Retrieve the Pre-Saved Trajectories . . . 55
5.3.4 Balance Set-Up . . . 57
5.3.5 Obtain the Crucible . . . 59
5.3.6 Do the Experiments . . . 60
5.3.7 Return the Crucible . . . 61
5.3.8 Unset the Balance . . . 62

6 Experimental Set-up **63**
6.1 Frame . . . 63
6.2 Crucibles . . . 65
6.3 Balance Device . . . 66
6.4 Ned2 Collaborative Robot . . . 66
6.4.1 Hardware Configurations . . . 66
Cobot's Gripper . . . 69
6.4.2 Software Tools . . . 69
ROS-Melodic . . . 70

6.5 Precision of Ned2 . . . 71
6.5.1 Required Precision . . . 71
6.5.2 Precision Validation . . . 71
6.6 Vibration Motor . . . 73

7 Evaluation & Results **75**
7.1 Evaluation & Conclusion of Experiment 1 . . . 75
7.1.1 Findings of Experiment 1 . . . 77
7.1.2 Conclusion of Experiment 1 . . . 78
7.2 Evaluation & Conclusion of Experiment 2 . . . 79
7.2.1 Findings of Experiment 2 . . . 81
7.2.2 Conclusion of Experiment 2 . . . 82
7.3 Results Summary . . . 82

8 Conclusion & Future Work **83**
8.1 Conclusion . . . 83
8.2 Future Discussions . . . 84

A GitHub Repository - Important Folders & Files **85**

B Examples of Tabular Data from the last Experiment **87**

C Important Python Fuctions Used in this Project **89**
C.1 Calling a Service Server . . . 89
C.1.1 Example of Call_Aservice Function . . . 90
C.2 Subscribing to Topic . . . 90
C.2.1 Examples of Subscribe Function . . . 90
C.3 Moveing the Cobot using Joints . . . 91
C.4 Moveing the Cobot using Pose . . . 92
C.5 Use Forward Kinematics . . . 94

D Balance Technical Drawing **95**

E Glass and Metal Crucibles Technical Drawing **99**

F Frame Technical Drawing **103**

Bibliography **109**

List of Figures

1.1 BORAMAT® Mono [60] 1

2.1 Examples of a humanoid and a collaborative robot. 6
2.2 Manipulators with 3 DoF of movement. 8
2.3 Illustration of different joints. [32] 8
2.4 The Wrist and End-effector Parts. [32] 9
2.5 How a Cobot range of motion and DoF are inspired by human motion. [37] 10
2.6 (P) Vector relative to frame A 11
2.7 All the transformers (TF) of Ned2 in Rviz. 13
2.8 Relationship between forward and inverse kinematics [17]. 14
2.9 High level diagram of various planning components 16
2.10 URDF diagram 17

3.1 Ubuntu file system structure 20
3.2 Graphical representation of a ROS system 23
3.3 Roscore Connections with the nodes 24
3.4 Files' structure for ROS workspace. [3] 25
3.5 Master Name Service Example [77] 28
3.6 Publisher–Subscriber communication model. 30
3.7 Server-Client Service communication 34
3.8 Server-Client Action communication 41
3.9 Choosing the kinematic solver for MoveIt planning group. 44

4.1 Examples of available powder dosing machines 45
4.2 Dependant and Independent Variables in our research 46

5.1 The sequence logic describing the full dosing process 53
5.2 Calibration process of Ned2 cobot. 54
5.3 The angle α with reference to the $x - axis$. 56
5.4 Defining the pre-named positions used in the trajectory planning. . . . 56
5.5 Indexing the crucibles' holders with numerical values. 56
5.6 Common flowchart elements [28] 57
5.7 Balance Setup flowchart 58
5.8 Obtain-Crucible flowchart 59
5.9 Do the Experiment flowchart 60
5.10 Return Crucible flowchart 61
5.11 Unset balance flowchart 62

6.1 The Full Framework of System 63
6.2 Demonstration of the parts of the framework. 64
6.3 The glass crucible with its hangers. 65
6.4 The metal crucible and its hanger. 65
6.5 The balance with its metal plates and rubber buffers. 66

6.6 Ned2 cobot provided by Niryo [22]. 66
6.8 The detailed workspace of Ned2 cobot. 67
6.7 The Servo motors used in Ned2. 67
6.9 The development phase of the gripper. 69
6.10 The main tasks of the gripper. 69
6.11 Software tools supported by Niryo in Ned2. [67] 70
6.12 The Catkin workspace in Ned2 70
6.13 The required precision for each task in our research 71
6.14 Recorded errors were measured along the x, y, and z axes for various velocities. 72
6.15 A way used to visualize the precision of the Ned2 cobot. 73
6.16 DO4 73

7.1 Manually fill the crucibles. 75
7.2 Illustration of dosing position. 76
7.3 Comparison between mass flow and dosed mass at full speed rotation ($1.775 rad/s$). 76
7.4 Comparison between mass flow and dosed mass at around 60% of maximum speed rotation ($1.0 rad/s$). 77
7.5 Comparison between mass flow and dosed mass at around 40% of maximum speed rotation ($0.71 rad/s$). 78
7.6 Division of dosing process [80]. 79
7.7 Comparison between mass flow and dosed mass at around 60% of maximum speed rotation ($1.0 rad/s$) to reach 1g of dosed mass. 81

A.1 A detailed explanation of the GitHub repository. 85

List of Tables

2.1 Robots Classification 7

3.1 ROS commands summary with some examples context. 43

5.1 Most important functions used in the research 52
5.2 The positions used to plan the trajectories. 55

6.1 Technical Specification of Ned2 [21] 68
6.2 The motor connections with the cobot. 73

8.1 Experiment summaries for the 4 different angles. 83

B.1 5^{th} Trial of the 2^{nd} experiment for angle 5°. 87
B.2 Last Trial of the 2^{nd} experiment for angle 13°. 88
B.3 5^{th} Trial of the 2^{nd} experiment for angle 16°. 88
B.4 Last Trial of the 2^{nd} experiment for angle 21°. 88

List of Abbreviations

API	**A**pplication **P**rogramming **I**nterface
BASH	**B**ourn **A**gain **SH**ell
Cobot	**C**ollaborative R**obot**
DH	**D**enavit-**H**artenberg
DoF	**D**egree **o**f **F**reedom
FK	**F**orward **K**inematics
GUI	**G**rafical **U**ser **I**nterface
ID	**D**ependent **V**ariable
IK	**I**nverse **K**inematics
IV	**I**ndependent **V**ariable
KDL	**K**inematics and **D**ynamics **L**ibrary
MPF	**M**otion **P**lanning **F**ramework
OpenCV	**Open** Source **C**omputer **V**ision Library
ROS	**R**obot **O**perating **S**ystem
TCP	**T**ool **C**enter **P**oint
TCP/IP	**T**ransmission **C**ontrol **P**rotocol/**I**nternet **P**rotocol
SSH	**S**ecure **Sh**ell
UML	**U**nified **M**odeling **L**anguage
URDF	**U**nified **R**obot **D**escription **F**ormat
TF	**T**rans**F**ormation

Chapter 1

Introduction

The primary objective of this thesis is to conduct an in-depth investigation into the flow characteristics of a powdered substance (Sugar) within a fully automated system, employing a collaborative robotic arm. Subsequently, following a comprehensive study of the observed behavior, we endeavor to discern the significance of individual parameters. *Our ultimate aim is to develop a precise* ***control algorithm based on these findings****, facilitating the precise dispensation of one gram of the respective powder.*

1.1 Motivation

FLUXANA® is firmly committed to providing comprehensive support to users engaged in X-ray fluorescence analysis (XRF) [8], a widely adopted spectroscopic method for elemental analysis. This methodology empowers precise identification of inorganic components within various substances and products. XRF analysis boasts extensive applicability across the spectrum of the quality assurance industry, as well as within institutions and regulatory bodies entrusted with overseeing compliance.

The precise dosing of micro quantities has been always a challenge in the industrial sector. Within FLUXANA®, this challenge is addressed through the utilization of the BORAMAT® as the powder dosing apparatus. The BORAMAT® Mono [60] and 18/30 [59] Material Doser represents an automated dosing system specifically designed for flux application in XRF analysis. Its purpose is to streamline laboratory operations, facilitating faster and more accurate weighing processes while offering real-time monitoring capabilities. Furthermore, its compatibility with standard laboratory scales ensures seamless integration into most laboratory environments without necessitating the procurement of new weighing equipment.

FIGURE 1.1: BORAMAT® Mono [60]

However, a limitation of the BORAMAT® system is its specialization in dosing XRF fusion samples. Moreover, certain applications of the BORAMAT® 18/30 system could potentially benefit from collaborative integration with another dosing system. In light of these considerations, FLUXANA® has embarked on this research endeavor. The objective is to investigate the impact of vibration, dosing angle, and rotational speed on the mass flow of powdered materials. A cobot with 6 DoF works together with an automated powder dosing system in order to carry out this study.

1.2 Objectives

The objectives of this research are summarised in the following:

- Insure a robust motion planning algorithm using ROS-melodic to do the experiments successfully.
- Experiment, observe, and analyze the behavior of the powder mass flow process, using different variations of various independent parameters.
- Use this well-investigated behavior to develop a precise control algorithm, facilitating the precise dosing of one gram of the respective powder.

This thesis does not aim to delve into the development of a custom robotic arm for experimentation purposes. Additionally, the scope excludes the creation of a machine-learning model from the data collected during the experiments. The limitations of the Ned2 cobot, particularly its constraints as a cost-effective alternative to industrial robots, are acknowledged. Furthermore, the project is constrained by available computational hardware and time limitations for the creation of a custom dataset. The primary focus remains on ensuring a robust motion planning algorithm, experimenting with powder mass flow variations, and developing a precise control algorithm for the targeted one-gram dosing objective.

1.3 Thesis Structure

This paper is divided into two parts and a conclusion. The first part, consisting of three chapters, provides the basic concepts necessary for understanding the experiments conducted in this research. We begin with an overview of robotics, including its components and motion planning frameworks. Then, we delve into the fundamentals of ROS to understand the logic employed in our system. Lastly, in the first part, we explore the powder dosage process and the parameters involved in the experiments as either dependent or independent variables.

Moving on to the second part, which also comprises three chapters, we discuss the methodology used in this research and explain the code in UML (Unified Modeling Language). In the subsequent chapter, we elaborate on the experimental setup for the research and the required components. Finally, we conclude the second part of the paper with the evaluation and results chapter. In the conclusion, we present our final thoughts on the results we have obtained. Additionally, we also provide concise recommendations for future research.

Part I

Basics of Robotics, ROS, and Powder Dosage

Chapter 2

Basics of Robotics

As an academic field, robotics emerges as a relatively youthful discipline, characterized by profoundly ambitious objectives, the most paramount of which is the creation of machines capable of emulating human behavior and cognitive processes. This quest to engineer intelligent machines inherently compels us to embark on a journey of self-exploration. It prompts us to scrutinize the intricacies of our own design—why our bodies possess the configurations they do, how our limbs synchronize in movement, and the mechanisms behind our acquisition and execution of intricate tasks. The realization that the fundamental inquiries in robotics are intrinsically linked to inquiries about our own existence forms a captivating and immersive aspect of the robotics pursuit. [41]

Definition

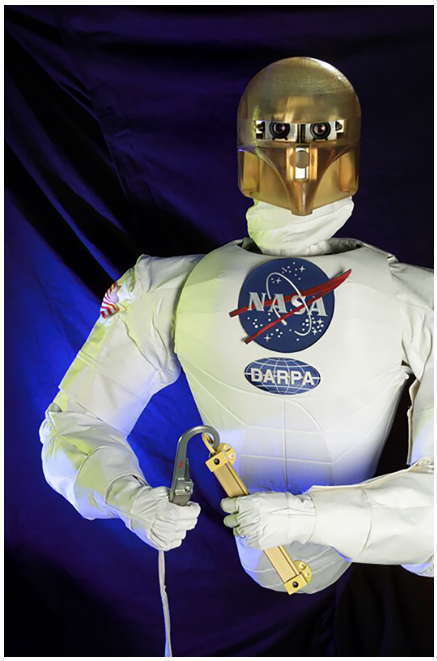
Robotics is the scientific field dedicated to the study of robots—machines capable of autonomous operation, carrying out various tasks without direct human intervention. [40]

While science fiction often envisions robots in humanoid or android forms, real-world robots, especially those designed for industrial applications, typically deviate from human physical resemblance. These robots typically comprise three fundamental components: a mechanical structure, often represented by a robotic arm, enabling physical interaction with the robot's environment or itself; sensors that collect data on various physical attributes such as sound, temperature, motion, and pressure; and a processing system that interprets data from the robot's sensors, providing instructions for task execution.

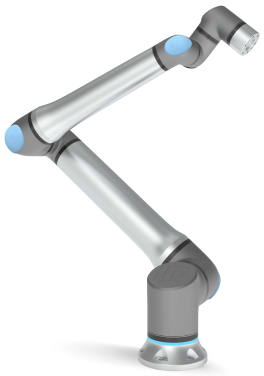
It's worth noting that certain devices, like web-crawling search engine bots that systematically explore the internet to collect information on links and online content, may lack physical mechanical elements. Nonetheless, they are still classified as robots because they exhibit the ability to perform repetitive tasks autonomously.

This chapter delves into an exploration of various robot classifications, delving into the foundational principles of mechanics and kinematics. It also scrutinizes the intricacies of planning and control within the context of collaborative robots (cobots).

The knowledge presented in this chapter draws significant inspiration from two primary sources, namely '**Modern Robotics**' [41] by Kevin M. Lynch and Frank C. Park, and '**Theory of Applied Robotics**' [32] authored by Professor Reza N. Jazar. For a deeper understanding of these topics, I encourage you to refer to these texts.



(a) NASA Robonaut [2] [7]



(b) Universal Robot (UR20) [72]

FIGURE 2.1: Figure (a) illustrates an instance of a humanoid robot developed by NASA, while Figure (b) exemplifies a cobot.

2.1 Types of Robots

Across diverse industries, robotics solutions have emerged as catalysts for heightened productivity, elevated safety standards, and increased operational adaptability. Organizations at the vanguard of innovation are discerning forward-looking applications of robotics that yield palpable and quantifiable outcomes. Intel collaborates closely with manufacturers, system integrators, and end-users, actively contributing to the realization of robots that deliver impactful, human-centered results.

According to an article from Intel regarding the classification of robots [30], the current generation of robots has been categorized into six distinct groups.

Autonomous Mobile Robots (AMRs) [18] AMRs navigate their environments and make rapid decisions on the fly. These robots employ advanced technologies like sensors and cameras to gather data from their surroundings. Equipped with onboard processing capabilities, they analyze this data and make well-informed decisions—whether it involves avoiding an approaching human worker, selecting the exact parcel to pick, or determining the suitable surface for disinfection. These robots are self-sufficient mobile solutions that operate with minimal human intervention. [61]

Automated Guided Vehicles (AGVs) [75] While AMRs navigate their surroundings autonomously, AGVs typically operate along fixed tracks or predetermined paths and frequently necessitate human supervision. AGVs find extensive application in scenarios involving the transportation of materials and goods within controlled settings like warehouses and manufacturing facilities.

Humanoids [38] While numerous mobile humanoid robots could, in a technical sense, be classified as Autonomous Mobile Robots (AMRs), this categorization

primarily applies to robots fulfilling human-centric roles, frequently adopting human-like appearances. These robots leverage a similar array of technological components as AMRs to perceive, strategize, and execute tasks, encompassing activities such as offering navigational assistance or providing concierge services.

Hybrids [64] Diverse categories of robots are frequently integrated to engineer hybrid solutions that possess the capacity to execute intricate operations. For instance, the fusion of an AMR with a robotic arm can yield a versatile system tailored for the handling of packages within a warehouse environment. As functionalities are amalgamated within single solutions, there is a concurrent consolidation of computational capabilities.

Articulated Robots [63] Commonly referred to as robotic arms, are designed to replicate the versatile functions of the human arm. These systems typically incorporate a range of rotary joints, varying from two to as many as ten. The inclusion of additional joints or axes equips these robotic arms with a wider range of motion capabilities, rendering them particularly well-suited for tasks such as arc welding, material manipulation, machine operation, and packaging.

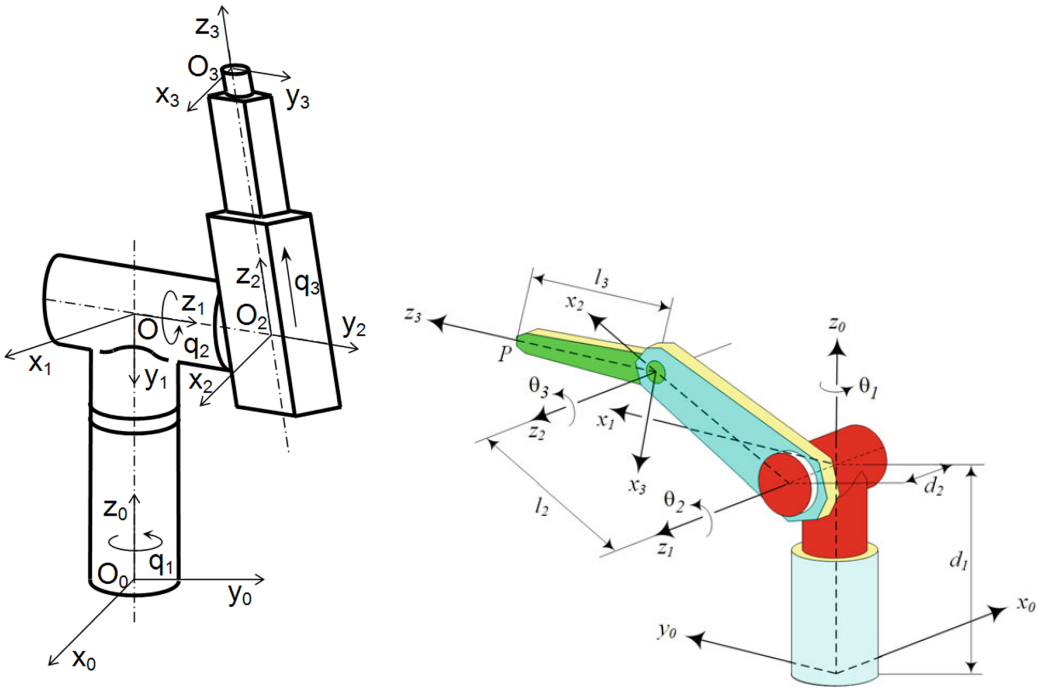
Cobots [51] Collaborative Robots, commonly referred to as cobots, are engineered with the specific purpose of working in tandem with, or directly alongside, human operators. Unlike many other categories of robots that function autonomously or within strictly segregated workspaces, cobots share work environments with human personnel to enhance their collective productivity. Their primary role often involves the removal of manual, hazardous, or physically demanding tasks from daily operations. In certain scenarios, cobots are capable of responding to and learning from human movements, further enhancing their adaptability.

The initial four robots fall under the category of mobile robots, possessing the capability to navigate within their surroundings, while the latter two are categorized as stationary robots, as detailed in table 2.1 below.

TABLE 2.1: Robots Classification.

Mobile	**Stationary**
AMRs	
AGVs	Articulated robots
Humanoids	**Cobots**
Hybrids	

Within the scope of this paper, our exclusive focus will be on **cobots** [51]. Across all the experiments conducted in this study, a cobot (Ned2, detailed and described in chapter 6) has been consistently utilized.



(a) 2R-1P manipulator. [74] (b) 3R manipulator. [32]

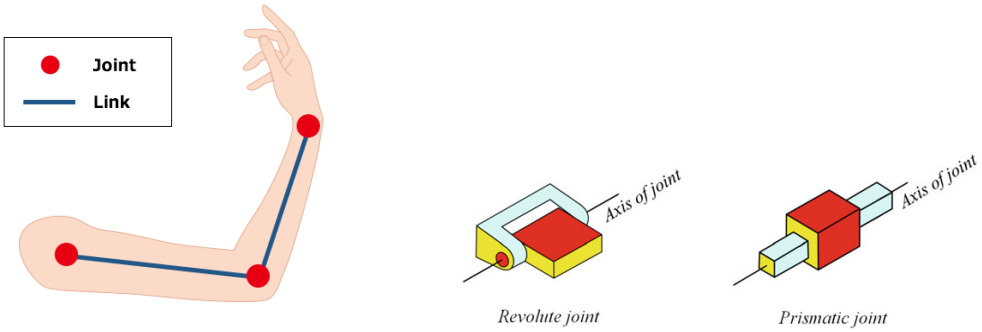
FIGURE 2.2: Manipulator with 3 DoF of movement.

2.2 Robot Components

In our study, we establish a kinematic model for a robotic manipulator, which is essentially a multi-body system comprising interconnected rigid bodies. These bodies are connected through revolute or prismatic joints (only revolute in our application), enabling relative movement. We employ principles of rigid body kinematics to elucidate the relative motions between these interconnected bodies.

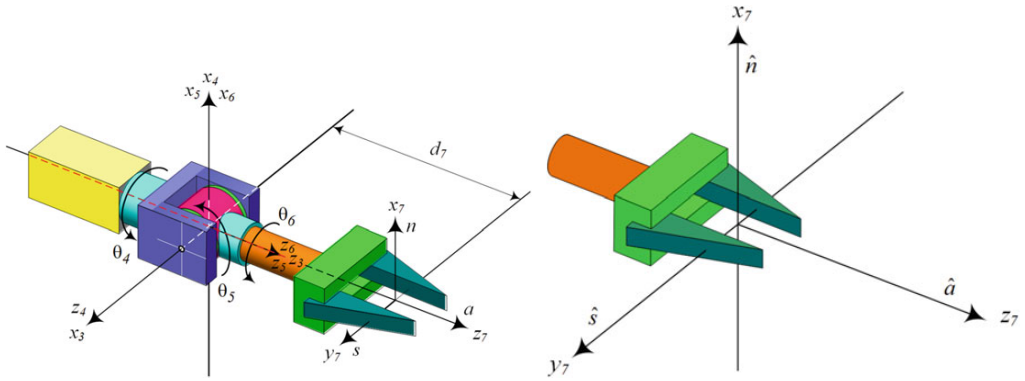
It's imperative to note that a comprehensive robotic system encompasses not only the manipulator or rover but also components such as the wrist, end-effector, actuators, sensors, controllers, processors, and software. [32]

Link: In the realm of robotics, each individual rigid component within a robot that



(a) A human elbow and shoulder are joints and the bones connecting them are links. [37] (b) Revolute and Prismatic joints.

FIGURE 2.3: Illustration of different joints.



(a) A spherical wrist kinematics. (b) End-effector.

FIGURE 2.4: The Wrist and End-effector. [32]

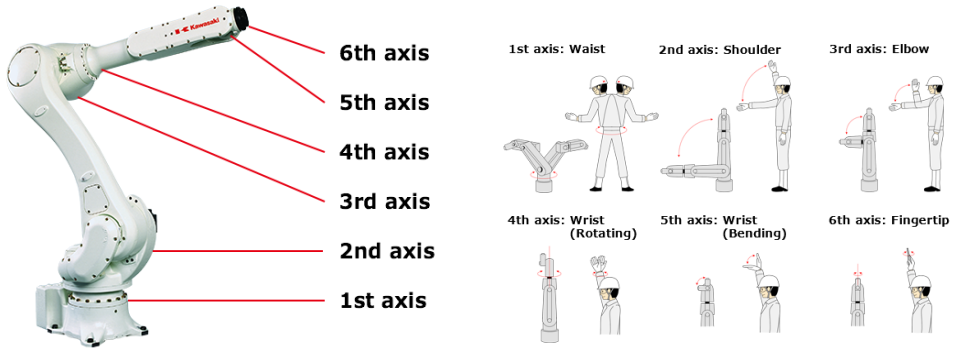
possesses the capacity to move concerning all other components is formally known as a 'link.' This terminology accommodates various descriptions, including 'bar,' 'arm,' or any object deemed equivalent to a link in the context of robot mechanics. A robot arm or link, in essence, represents a solid, rigid element capable of relative motion when compared to the other links within the robotic structure.
Moreover, when we encounter two or more linked components that are entirely constrained in terms of relative movement, they are collectively regarded as a 'compound link,' forming a unified and motionally inseparable entity within the robot's framework.

Joint: A kinematic pair refers to two rigid bodies that are in constant contact with one another, enabling potential relative motion. These pairs connect two links at a joint where their relative movement is described by a single joint coordinate. Typically, joints manifest as revolute (rotary) or prismatic (translatory). In Figure 2.3(a), the illustration portrays joints and links within a human arm. A revolute joint (R) mimics a hinge, enabling relative rotation between two links, while a prismatic joint (P) facilitates translational relative motion between the two links.

The relative rotation of interconnected links through a revolute joint transpires along an axis referred to as the joint axis. Similarly, the translation between two connected links via a prismatic joint occurs along an axis, also known as the joint axis. The joint coordinate, or joint variable, represents the value describing the relative position of these connected links at the joint. For a revolute joint, it represents an angle, while for a prismatic joint, it denotes a distance. The depiction of revolute and prismatic joints is respectively shown in Figures 2.3(b).

Manipulator: The primary structure of a robot comprises links, joints, and associated structural components, known as the manipulator. A manipulator is transformed into a robot when integrated with a wrist, gripper, and its control system. However, in the literature, robots and manipulators are often used interchangeably, both denoting robotic systems. In Figure 2.2(b), a 3R manipulator is depicted, while Figure 2.2(a) showcases a 2R-1P manipulator arm.

Wrist: The joints situated in the kinematic chain between the forearm and the end-effector are termed the wrist. It's customary to design manipulators with



(a) Kawasaki Cobot.

(b) The respective motion of each axis compared to human motion.

FIGURE 2.5: How a Cobot range of motion and DoF are inspired by human motion. [37]

spherical wrists, comprising three intersecting revolute joint axes that converge at a common point known as the wrist point. Illustrated in Figure 2.4(a) is a stationary spherical wrist composed of three revolute joints with orthogonal rotation axes. The spherical wrist significantly streamlines kinematic analysis by separating the end-effector's positioning and orientation. Consequently, the manipulator's wrist point holds three degrees of freedom for positioning, controlled by three arm joints. The orientation's degrees of freedom depend on the wrist design, which may involve one, two, or three degrees based on the specific application.

End-Effector: The end-effector constitutes the segment attached to the final link that carries out the designated tasks of the robot. Typically, the simplest end-effector is a gripper, allowing two actions: opening and closing. Both the arm and wrist assemblies primarily facilitate the positioning of the end-effector and any accompanying tools. It is the end-effector that executes the intended tasks. Considerable research is dedicated to developing specialized end-effectors and tools, including anthropomorphic hands designed for prosthetic purposes in manufacturing. Consequently, a robot is comprised of a manipulator or mainframe, a wrist, and an end-effector, while the wrist and end-effector assembly is sometimes referred to as a hand. Figure 2.4 presents an example of an end-effector.

Actuator: Actuators serve as the driving force akin to a robot's muscles, enabling the alteration of their configuration. They supply the necessary power to act on the mechanical structure, counteracting external forces like gravity and inertia, to adjust the geometric position and orientation of the robot's hand. Actuators, typically electric, hydraulic, or pneumatic, must be controllable.

Sensors: Sensors are components employed to observe and gather data related to both internal and external conditions. Within the context of this book, pivotal data such as joint positions, velocities, accelerations, and forces hold significant importance for measurement. Embedded within the robot system, these sensors relay pertinent details concerning each link and joint to the control unit. The control unit, based on this information, ascertains the robot's configuration.

Controller: The robot's controller, or control unit, fulfills three primary functions:

1. Information role, involved in the collection and processing of data obtained from the robot's sensors.
2. Decision role, responsible for planning the geometric movement of the robot structure.
3. Communication role, tasked with managing information exchange between the robot and its environment. The control unit encompasses both the processor and software components.

2.3 Spatial descriptions and transformations

The basic concept of robotic manipulation is the movement of various components and tools utilizing a mechanism of some sort. In order to achieve this, it's necessary to have a system for expressing the positions and orientations of these components and mechanisms. To accomplish this, the introduction of a coordinate system is required, as well as the development of representation conventions. These concepts will serve as a framework for exploring linear and rotational velocities, forces, and torques in future discussions. [14]

2.3.1 Position, Orientation, and Frames

When interacting with different components within a system of manipulation, a description is implemented to identify their specific characteristics. These elements include the manipulator itself, alongside the parts and tools involved. The aspect of positions, orientations, and a combination of both - referred to as a frame - are all explored in detail within this section.

Position

Definition

With an established coordinate system, it's possible to locate any point in the universe using a **position vector** of 3 x 1.

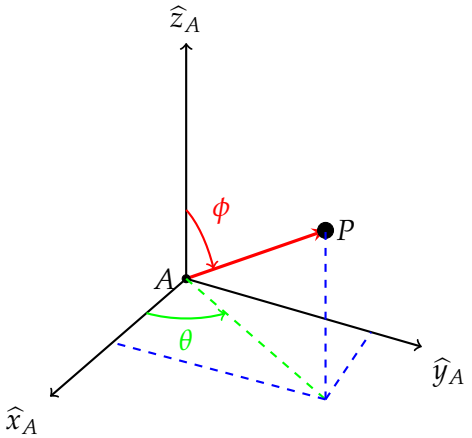


FIGURE 2.6: (P) Vector relative to frame A

As we tend to define multiple coordinate systems besides the universe coordinate system, it becomes important to attach information to each vector so as to pinpoint the particular coordinate system it belongs to. For instance the example shown in Fig. 2.6, the position of point P is described relative to coordinate frame A, and written is the following notation: ${}^{A}P$.
The position ${}^{A}P$ could be represented as a vector with 3 components:

$$ {}^{A}P = \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} \tag{2.1} $$

Orientation

The position of a point is defined in relation to a specific coordinate system. Similarly, the orientation of a point is described with respect to another frame or coordinate system. The orientation of a robot is determined by a combination of rotations in the X, Y, and Z directions. Ned2 Cobots as well as Universal Robots utilize the axis-angle representation to describe the orientation of the robot. This representation involves a direction vector around which the orientation can be rotated by a specific angle, theta. By performing rotations, the orientation of the robot can be modified. [57]

Frame

A frame in robotics defines the coordinate system of a robot, allowing it to determine its position in space and the locations of relevant objects or areas relative to the robot. Robotics use three main coordinate frames: the WORLD frame, the USER (Base) frame, and the TOOL frame. The WORLD frame is a fixed cartesian coordinate frame that represents the center point at the base of the robot. The USER (Base) frame is a cartesian coordinate system that can be used to define a workpiece or an area of operation. The TOOL frame is a cartesian coordinate system that specifies the location and orientation of the tool. [27]

2.3.2 TransFormers

In the field of robotics, transformers are commonly used mathematical transformations that convert coordinates between different frames of reference. Transformation matrices are essential for describing the position and orientation of a robot's end-effector in relation to its base or previous joint. They play a crucial role in implementing both forward and inverse kinematics.

However, Google Research has also introduced a concept called "Robotics Transformer" (RT-1) [25]. The RT-1 is a multi-task model that tokenizes inputs from robots and generates corresponding actions, such as camera images, task instructions, and motor commands. This model enables efficient inference during runtime, making real-time control possible. For our used cobot, we only use the classical transformation matrices.

2.3.3 Jacobian Matrix

In robotics, the Jacobian matrix is a mathematical tool used to establish the relationship between the joint velocities and the end-effector velocities of a robot manipulator. The joint velocities in a robotic arm are transformed into the end effector velocity

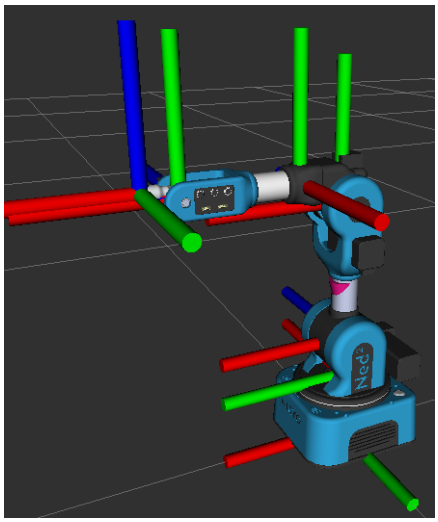


FIGURE 2.7: All the transformers (TF) of Ned2 in Rviz.

through a facilitating process known as a conversion of angular velocities. For instance, when the servo motors of a robotic arm are rotating at a certain velocity (e.g., in radians per second), the Jacobian matrix can be employed to calculate the speed at which the end effector of a robotic arm is moving, encompassing both linear velocity (x, y, z) and angular velocity (`roll` ωx, `pitch` ωy, and `yaw` ωz). The Jacobian matrix, represented as J, converts the velocities of the joints into velocities of the end effector according to the following equation [1]:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}_{6\times 1} = \begin{bmatrix} J_{11} & J_{12} & \cdots & \cdot & \cdot & J_{1n} \\ J_{21} & J_{22} & \cdot & \cdot & \cdot & J_{2n} \\ \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\ J_{61} & \cdots & \cdot & \cdot & \cdot & J_{6n} \end{bmatrix}_{6\times n} \times \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \vdots \\ \dot{q}_n \end{bmatrix}_{n\times 1} \tag{2.2}$$

Where:

- The symbol q with a dot on top represents the joint velocities, indicating the speed of rotation for a revolute joint and the speed of extension or contraction for a prismatic joint. This vector is of size $n \times 1$, with n representing the number of joints (e.g., servo motors or linear actuators) in your robotic arm.
- The variables x, y, and z with dots on top denote linear velocities, which indicate the speed at which the end effector is moving in the x, y, and z directions relative to the base frame of a robotic arm.
- The angular velocities ωx, ωy, and ωz represent the speed at which the end effector rotates around the x, y, and z axes of the robotic arm's base frame. [1]

The Jacobian matrix J has m rows and n columns, where m is 3 in two dimensions and 6 in the context of three-dimensional robots. These values correspond to the number of joints. [41]

The Jacobian matrix is composed of two parts: the Linear Velocity Jacobian (Jv) and the Angular Velocity Jacobian ($J\omega$). The upper part pertains to the linear velocities of the end-effector, while the lower part pertains to the angular velocities resulting from changes in all joint velocities.

The Jacobian matrix of a robotic manipulator is derived using different methods for Jv and $J\omega$. These are found separately and then combined to form the final Jacobian matrix. For Jv, we differentiate the position functions for the x, y, and z coordinates of the end-effector with respect to the joint variables $[q_1, q_2, q_3, ..., q_n]$ to obtain Jv, which is related to the linear velocities of the end-effector due to joint velocities.

2.4 Forward & Inverse Kinematics

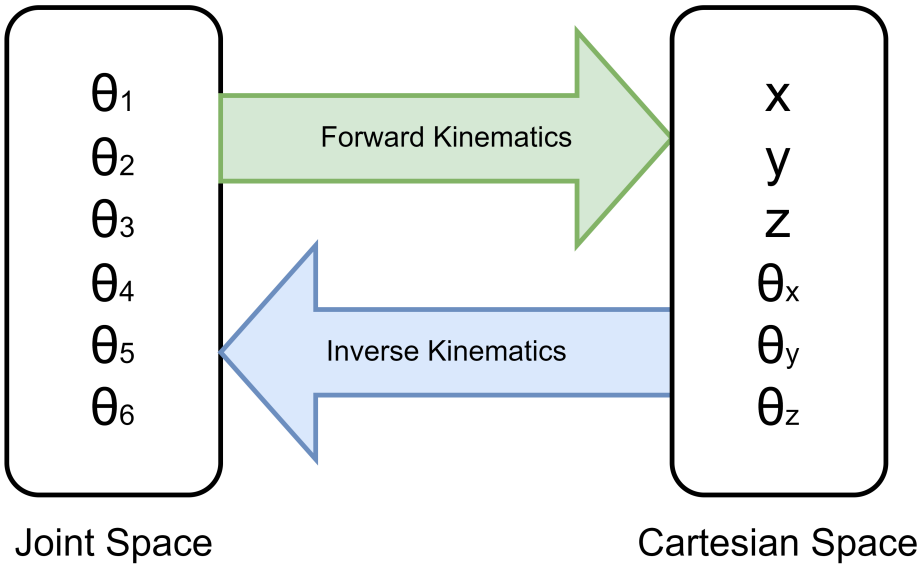


FIGURE 2.8: Relationship between forward and inverse kinematics [17].

As depicted in Figure 2.9, forward and inverse kinematics are essential components of motion planning algorithms.

2.4.1 Forward Kinematics

Definition

Forward kinematics involves using the robot's kinematic equations to calculate the position of the end-effector based on specified values for the joint parameters.

It focuses on predicting the final state of a system given the initial conditions and inputs [48].

When a kinematic chain is made up of links and joints with multiple degrees of freedom, forward kinematics can determine the position and orientation of the end-effector in the operational workspace, provided that all the joint parameters are known [43]. The kinematic equations for a robot's series of links are obtained by using a rigid transformation Z to represent the relative motion allowed at each joint, and a separate rigid transformation X to define the dimensions of each link. This results in a sequence of alternating rigid transformations, including joint and link transformations from the base of the chain to its final link.

The kinematic equations for a serial chain consisting of n links, with joint parameters θ_i, can be represented as:

$$[T] = [Z_1][X_1][Z_2][X_2] \cdots [Z_n][X_n] \tag{2.3}$$

The transformation matrix T represents the frame-to-frame transformation in robotics. This transformation is typically described using Denavit-Hartenberg parameters [43]. You can refer to the research paper by Danavit and Hartenberg for detailed information on the Denavit-Hartenberg parameters. Alternatively, you can also check out the simplified article from Universal Robots for a more accessible explanation.

2.4.2 Inverse Kinematics

Definition

> **Inverse kinematics** is the reciprocal computation of forward kinematics, entailing the determination of variable joint parameters essential for situating the terminus of a kinematic chain, such as a robot manipulator, in a specified position and orientation concerning the chain's origin. [52][41]

In the realm of robotics, inverse kinematics leverages kinematics equations to ascertain the joint parameters that yield a desired configuration, encompassing position, and rotation, for each of the robot's end-effectors. This holds significance as robotic tasks hinge on the manipulation of end effectors, while the exertion of control pertains to the joints [48].
The process of ascertaining the robot's movement to transition its end-effectors from an initial configuration to a target configuration is denoted as motion planning [41]. Inverse kinematics serves to translate the motion plan into joint actuator trajectories, dictating the robot's movement.
Inverse kinematics is a mathematical procedure employed to compute the joint positions necessary to situate a robot's end effector at a specified position and orientation, often referred to as its "pose". A dependable inverse kinematic solution is imperative for programming a robot to execute designated tasks.

In the realm of inverse kinematics, multiple solutions and diverse approaches to computing the inverse kinematic solution are prevalent [4]. Inverse kinematics algorithms meticulously determine the precise position of each joint in the robot, requisite for achieving the desired end effector pose. Please refer to the comprehensive survey on Inverse Kinematics Techniques in Computer Graphics by Aristidou (2018) [4].

2.5 Motion Planning

The realm of robotic motion planning has emerged as an integral and evolving discipline in the field of robotics [44]. Its focus is on the systematic transformation of task directives into isolated motion sequences that allow machines to navigate environments confining them.
The software development of a motion planning framework (MPF) is challenging and involves combining many disparate fields of robotics and software engineering [50]. In this context, the software is labeled as a framework because of its ability

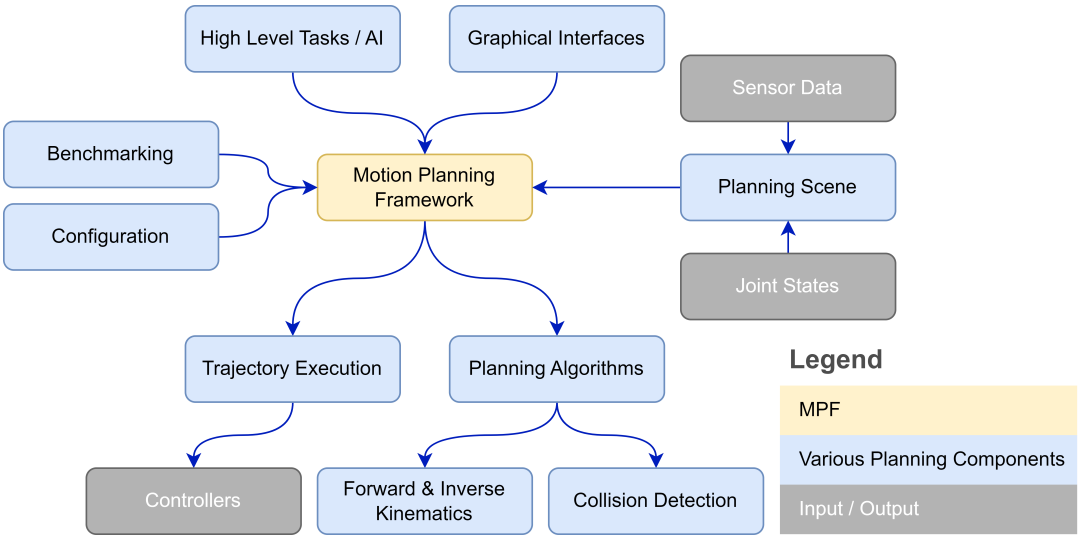


FIGURE 2.9: High level diagram of various planning components in a Motion Planning Framework (MPF). [12]

to generalize the components of motion planning through interfaces, which we will further examine later on.

2.5.1 Motion Planning Framework (MPF)

A variety of software components, known as planning components, are necessary for an MPF to function along with common data structures. The planning components are illustrated in Figure 2.9, and involve numerous interactions. Among them is a planning component that carries out motion planning. It encompasses one or several algorithms suitable for resolving the challenges met by a robot. Although the scope of motion planning is vast, there is no unified formula catering to all situations. As such, a framework free of robot constraints should comprise different algorithms as well as their variants.

Several important aspects must be considered in robot planning. First, a collision-checking module is necessary to identify possible intersections of geometric objects in the planning environment and the robot model. Additionally, a forward kinematics solver is essential for establishing the robot's geometry based on joint positions, while an inverse kinematics solver becomes necessary when some planning approaches rely on Cartesian end effector space. Different constraints, such as joint/velocity/torque limits and stability requirements, may also require other components.

To create a truly effective MPF, the inclusion of supplementary elements is essential. Upon reaching a configuration space solution, articulated through position waypoints, a corresponding timeframe-based trajectory max needs to be integrated for proper execution. At runtime, a controller manager selects the appropriate joint-related controllers from its available resources for any given path parameter of a trajectory. Alongside the recognition data gathered through its priority perception pipeline, integrated secondary components in MPF-assembled object ownership need to augment object perception awareness and provide additional diagnostics data. Meanwhile, application-specific taskings like pick-and-place ultimately yield the above by taking into account the respective contributions of said motion-planning string instruments. Additional encapsulated MOF behaviors available as

add-ons include debugging and analysis tools, widespread testing suites, as well as the temptation of the discernible user screen perception. [12]

2.5.2 URDF

Definition

Robot Model Format: The robot model in robotics serves as a structured representation encompassing a robot's three-dimensional design, kinematics, and various properties, such as geometric visualization meshes, collision geometry for efficient collision checking, joint limits, sensors, and dynamic characteristics like mass, moments of inertia, and velocity limits.

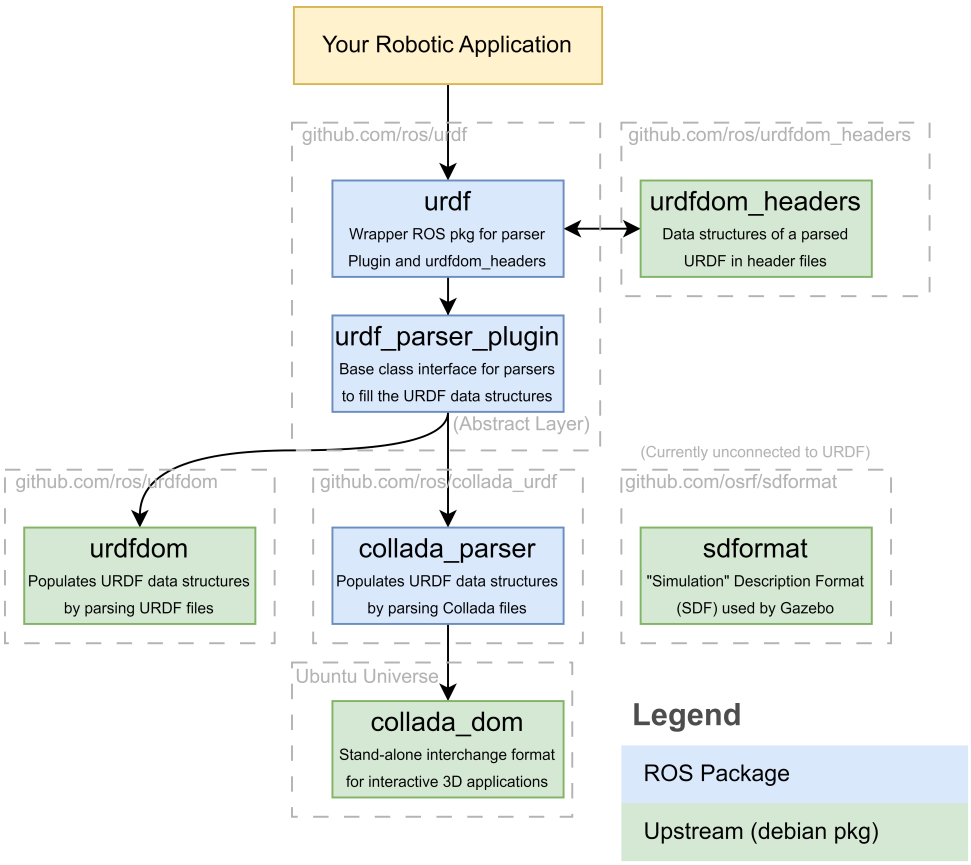


FIGURE 2.10: A diagram that attempts to explain how a number of different packages and components make up URDF [49]

While the typical representation involves a kinematic tree illustrating relationships between joints and links, this approach encounters challenges with robots featuring closed chains. For the purposes of our application and many modern Motion Planning Frameworks (MPFs), we define modeled robots as arbitrarily articulated rigid bodies.

Effective robotics software demands a standardized format capable of expressing diverse hardware configurations while being user-friendly for setup. In MoveIt! (a ROS motion planning framework, explained in section 3.10) for example, this requirement is met by adopting the **Unified Robotic Description Format** (**URDF** [49]) Document Object Model. This data structure is populated through the interpretation

of human-readable XML schemas, including URDF-formatted files (distinct from the data structure) and the industry-standard Collada [31] format.

Creating a precise robot model can be a challenging task. While URDF models for many robots are readily available, facilitating user avoidance of this challenge, custom robots necessitate the development of a new robot model. In such cases, the URDF model in ROS proves to be the most suitable choice, providing users with tools for XML validation, visualization, and the direct conversion of a SolidWorks CAD model into URDF format [6].

URDF Components in Hydro

- The core URDF parser and data structures (urdfdom, urdfdom_headers) are now stand-alone packages.
- These packages are separated from ROS (check next chapter 3 for more details) dependencies and are slated for independent release into Ubuntu.
- A clear distinction exists between a URDF file and a URDF data structure:
  - A URDF file adheres to the XML format outlined on the `http://wiki.ros.org/urdf/XML`.
  - A URDF data structure comprises a set of versatile classes for parsing various formats (currently URDF and Collada).
- The introduction of a new plugin abstraction layer (urdf_parser_plugin) facilitates populating URDF data structures with different file formats, presently supporting URDF and Collada.
- ROS packages urdf_parser and urdf_interface have been deprecated in Groovy and removed in Hydro, reflecting ongoing optimizations in URDF handling within ROS.

2.6 Conclusion

In conclusion, motion planning is a pivotal aspect of the field of robotics. It enables precise control over a robot's movements by calculating the necessary joint configurations for desired end-effector poses, or vice versa for forward kinematics. As we transition to the upcoming chapter on the Robot Operating System (ROS), it's crucial to recognize how ROS leverages such foundational principles, providing a comprehensive framework for the development and control of robotic systems.

Chapter 3

Robot Operating System | ROS

Definition

ROS serves as an open-source, meta-operating system designed to cater to the requirements of your robot. Much like traditional operating systems, it offers essential functionalities such as hardware abstraction, precise control over low-level devices, integration of commonly-used features, seamless message exchange between processes, and streamlined package management. Furthermore, ROS equips you with a comprehensive set of tools and libraries to facilitate tasks like code acquisition, compilation, development, and execution across diverse computing environments. [56]

In this chapter, we will delve into the foundational elements of the ROS framework. We will begin by exploring the relevance and basics of Linux and Ubuntu in the context of ROS. Following that, we will delve into the philosophical underpinnings of ROS, its master communication structure, and how these elements relate to topics, services, and actions. Finally, we will provide an explanation of the motion planning framework known as MoveIt.
All the code samples used in the chapter to explain ROS main concepts are either written by me, taken from the referenced books and articles, or cited from the Construct platform for robotics.

3.1 Linux for Robotics

Linux is a free, open-source operating system that includes several utilities that will significantly simplify your life as a robot programmer. As will be shown in the upcoming sections, ROS (Robot Operating System) is based on a Linux system. All commands and concepts explained here are taken from the Linux tutorial made by the University of Surrey. [35]

3.1.1 What Is Ubuntu? and Why for Robotics?

Ubuntu, accessible at `www.ubuntu.com`, stands as a widely acclaimed Linux distribution rooted in the Debian architecture (source: `https://en.wikipedia.org/wiki/Debian`). Notably, it's freely available and open source, permitting extensive customization for specific applications. Ubuntu boasts an extensive software repository, comprising over 1,000 software components, encompassing essentials such as the Linux kernel, GNOME/KDE desktop environments, and a suite of standard desktop applications, including word processing tools, web browsers, spreadsheets, web servers, programming languages, integrated development environments (IDEs), and

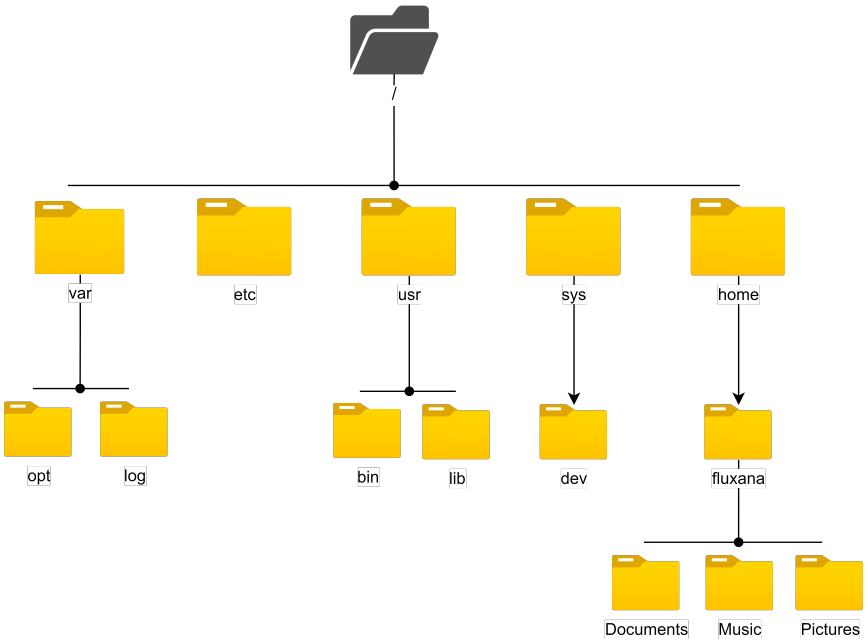


FIGURE 3.1: Ubuntu file system structure

even PC games. Versatile in its deployment, Ubuntu can operate on both desktop and server platforms, accommodating architectures like Intel x86, AMD-64, ARMv7, and ARMv8 (ARM64). Canonical Ltd., headquartered in the UK (www.canonical.com), provides substantial backing to Ubuntu. [39]

In the realm of robotics, software stands as the nucleus of any robotic system. An operating system serves as the foundation, facilitating seamless interaction with robot actuators and sensors. A Linux-based operating system, such as Ubuntu, offers unparalleled flexibility in interfacing with low-level hardware while affording provisions for tailored OS configurations tailored to specific robot applications. Ubuntu's merits in this context are manifold: it exhibits responsiveness, maintains a lightweight profile, and upholds stringent security measures. Additionally, Ubuntu boasts a robust community support ecosystem and a cadence of frequent releases, ensuring its perpetual relevance. It also offers long-term support (LTS) releases, guaranteeing user assistance for up to five years. These compelling attributes have cemented Ubuntu as the preferred choice among developers in the Robot Operating System (ROS) community. Indeed, Ubuntu stands as the sole operating system that enjoys comprehensive support from ROS developers. The Ubuntu-ROS synergy emerges as the quintessential choice for programming robots. [35]

3.1.2 Ubuntu File Structure

Similar to the 'C drive' in a Windows operating system, Linux incorporates a dedicated storage area for its system files, known as the root file system. This root file system is established during the Ubuntu installation process, with the assignment of '/' as its designated mount point. For a visual representation of the Ubuntu file system architecture, refer to Figure 3.1.

The following describes the uses of each folder in the file system:

- */bin and /sbin*: These directories house essential system applications, akin to the 'C:/Windows' folder in Windows.

- */etc*: Within this directory, system configuration files are stored.
- */home/yourusername*: Equivalent to the 'C:/Users' directory in Windows, this directory serves as the user's home.
- */lib*: Similar to '.dll' files in Windows, the '/lib' directory contains library files.
- */media*: This directory serves as the mount point for removable media.
- */root*: The '/root' directory contains files associated with the root user, who holds administrative privileges in the Linux system.
- */usr*: Pronounced 'user,' the '/usr' directory hosts a majority of program files, akin to 'C:/Program Files' in Microsoft Windows.
- */var/log*: Within this directory, you'll find log files generated by various applications.
- */home/yourusername/Desktop*: The location for Ubuntu desktop files.
- */mnt*: Mounted partitions are accessible in this directory.
- */boot*: This directory stores essential files required for the boot process.
- */dev*: Linux device files are located here.
- */opt*: The '/opt' directory serves as the designated location for optionally installed programs. (For instance, ROS is installed in '/opt').
- */sys*: This directory houses files containing critical information about the system.

3.2 Philosophy Behind ROS

The philosophical objectives of ROS can be succinctly described as follows [54]:

- Decentralized collaboration: Emphasizing **peer-to-peer** interactions. ROS consists of multiple interconnected software components enabling continuous message exchange without centralized routing, allowing for scalability even with increased data volume, although it may add complexity to the system infrastructure.
- **Tool-oriented** approach: Focusing on the development of a robust set of tools. ROS, influenced by Unix's architectural principles, constructs complex software systems through multiple small, versatile programs, differentiating itself from other robotics software by decentralizing various tasks to discrete software, enabling continual evolution and tool enhancement for specific task domains.
- **Multilingual support**: Enabling compatibility with multiple programming languages. ROS embraces a multilingual approach, allowing modules to be written in diverse programming languages, including C++, Python, LISP, Java, JavaScript, MATLAB [13], Ruby, Haskell, R, Julia, and others, with this book predominantly using the Python client library for code examples, yet noting the flexibility to utilize other available client libraries for discussed tasks.

- **Thin** design: Prioritizing a streamlined framework. ROS conventions encourage an approach where developers build standalone libraries that can be incorporated into ROS modules, fostering versatile software reuse and enabling streamlined automated testing via established continuous integration tools.
- Openness and freedom: Being freely available and based on **open-source** principles. ROS, operating under the permissive BSD license [55], enables diverse licensing arrangements by allowing flexible integration of closed-source and open-source modules, accommodating commercial, academic, and hobby projects, ensuring compliance within varied environments [54].

To the best of our knowledge, no existing framework encompasses this specific set of design principles. This section aims to delve into these philosophies, elucidating how they have profoundly influenced the design and implementation of ROS [54].

3.3 Preliminaries

Before delving into ROS, it's essential to introduce the fundamental concepts that underpin this framework. ROS systems are constituted by a multitude of autonomous programs that maintain continuous communication with one another. This section provides an in-depth exploration of this architectural setup and the associated command-line tools. It further delves into the intricate aspects of ROS naming conventions and namespaces, demonstrating their role in facilitating code reusability. [53]

3.3.1 ROS-Graph [11]

One of the original challenges inspiring the creation of ROS was commonly known as the 'fetch an item' problem. This scenario involved a relatively large and complex robot equipped with various sensors, a manipulator arm, and a mobile base. In the 'fetch an item' problem, the robot's objective is to navigate a typical home or office environment, locate a specified item, and transport it to the designated location. This task led to several key observations, which subsequently became foundational design goals for ROS:

- The application task can be broken down into numerous autonomous subsystems, encompassing areas like navigation, computer vision, and grasping.
- These subsystems are adaptable for various tasks, such as security patrols, cleaning, and mail delivery, among others.
- By implementing appropriate hardware and geometry abstraction layers, the majority of application software can be made compatible with different robotic platforms.

These principles are exemplified through the core structure of a ROS system: its graphical representation. In ROS, multiple programs operate concurrently and communicate by exchanging messages. This system structure is conveniently portrayed as a mathematical graph, with nodes representing individual programs and edges indicating their communication. While Figure 3.2 illustrates a sample ROS graph from one of the early 'fetch an item' implementations, the specific details are less significant compared to the overarching concept of a ROS system as an assembly of

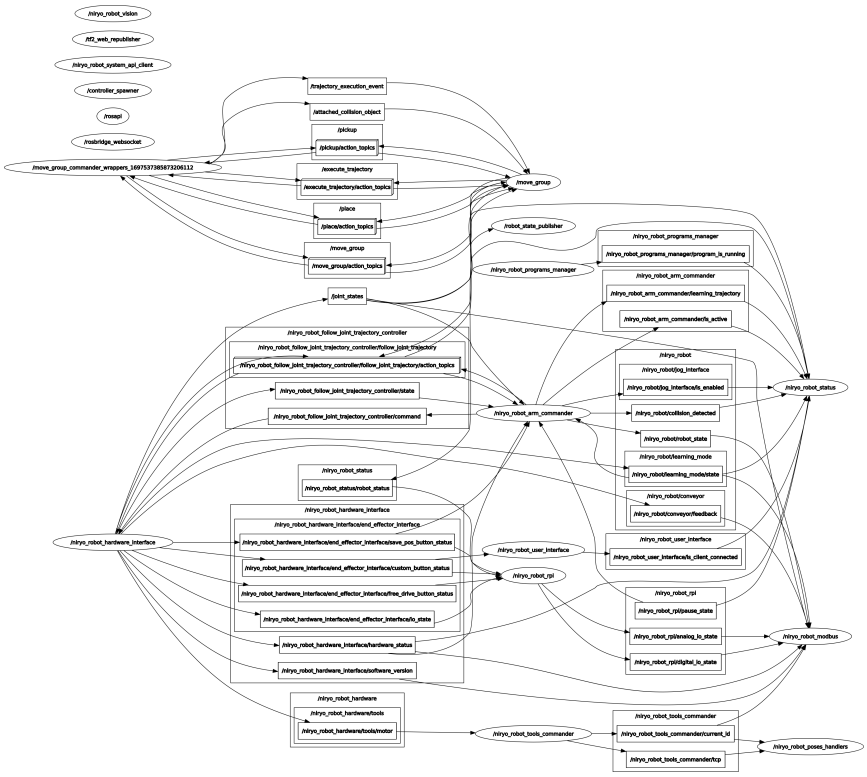


FIGURE 3.2: Graphical representation of a ROS system for 'Niryo-Ned2 [21]' robot—nodes/topics within the graph symbolize individual software modules, while edges denote message streams facilitating the exchange of sensor data, actuator commands, planner states, intermediate representations, and other relevant information.

nodes engaged in message-based communication. This representation serves as a practical framework for software development, emphasizing the modular nature of ROS programs, or 'nodes,' as integral components within a larger system.

In summary, within a ROS graph, a node signifies a software module engaged in message transmission, and an edge denotes the flow of messages between two nodes. While complexity can increase, nodes are typically POSIX processes, and edges are akin to TCP connections, enhancing fault tolerance as a software crash typically affects only the crashing process, leaving the rest of the graph operational. The events leading to the crash can often be reconstructed by logging messages entering a node and replaying them within a debugger at a later time.

One of the most significant advantages of a loosely coupled, graph-based architecture is the capacity to rapidly prototype complex systems with minimal or no need for additional 'glue' software during experimentation. Individual nodes, such as the object recognition node in a 'fetch an item' system, can be effortlessly replaced by launching an entirely different process that handles images and generates labeled objects. Beyond node replacement, entire segments of the graph (subgraphs) can be dynamically dismantled and substituted with other subgraphs in real time. This flexibility extends to replacing real-robot hardware drivers with simulators, swapping navigation subsystems, fine-tuning algorithms, and more. Since ROS dynamically generates the necessary network backends, the entire system fosters an interactive environment that encourages experimentation.

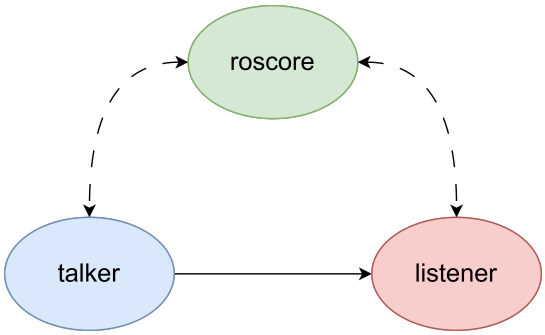


FIGURE 3.3: `roscore` establishes ephemeral connections with the other nodes in the system.

To this point, we have assumed that nodes discover each other, but we have not elaborated on the process. Amidst the extensive network traffic, how do nodes locate and initiate message exchange? The solution lies in a program known as '`roscore`'.

3.3.2 Roscore [76]

`roscore` serves as a vital component within the ROS ecosystem by facilitating connections between nodes to enable message transmission. During initialization, each node registers its published message streams and desired subscriptions with `roscore`, allowing it to establish direct peer-to-peer connections with other nodes participating in the same message topics. A functioning `roscore` is imperative for any ROS system since it serves as a vital reference point for nodes to discover one another.

It's important to note that while `roscore` plays a crucial role in aiding nodes in locating their peers, the actual message transmission between nodes occurs in a peer-to-peer manner. This setup can sometimes be misconstrued, especially for individuals accustomed to client/server systems from web-based backgrounds, wherein the roles of clients and servers are more distinct. The ROS architecture, however, functions as a hybrid system, integrating aspects of both client/server and fully distributed models, thanks to the central role of `roscore`, which acts as a naming service for peer-to-peer message streams.

When a ROS node initiates, it relies on the presence of an environment variable, ROS_MASTER_URI, which should contain a URL of the form `http://hostname:11311/`. This URL signifies the existence of a functioning `roscore` accessible on port 11311, hosted on a machine named hostname, which can be reached over the network.

With this information, nodes communicate with `roscore` at startup to register themselves and query for other nodes and message streams by name. Each node informs `roscore` about the messages it can provide and those it wishes to subscribe to. `roscore`, in return, supplies the necessary details about the message producers and consumers. In graphical terms, every node within the system can periodically utilize `roscore` services to identify and connect with its peers. This is illustrated by the dashed lines in Figure 3.3, signifying that in a basic two-node setup, the talker and listener nodes intermittently make service calls to `roscore` while directly engaging in peer-to-peer message exchange.

`Roscore` also serves as a parameter server, extensively utilized by ROS nodes for

configuration purposes. It enables nodes to store and retrieve various data structures, including robot descriptions and algorithm parameters. To interact with the parameter server, ROS provides a command-line tool called 'rosparam,' which we will use throughout this book.

We will delve into examples of using roscore shortly. For now, it's essential to remember that roscore facilitates nodes in discovering other nodes. Before we proceed to run some nodes, it's worth understanding how ROS organizes packages and gaining some insight into the ROS build system, known as 'catkin. [53]

3.3.3 catkin, Workspaces, and ROS Packages

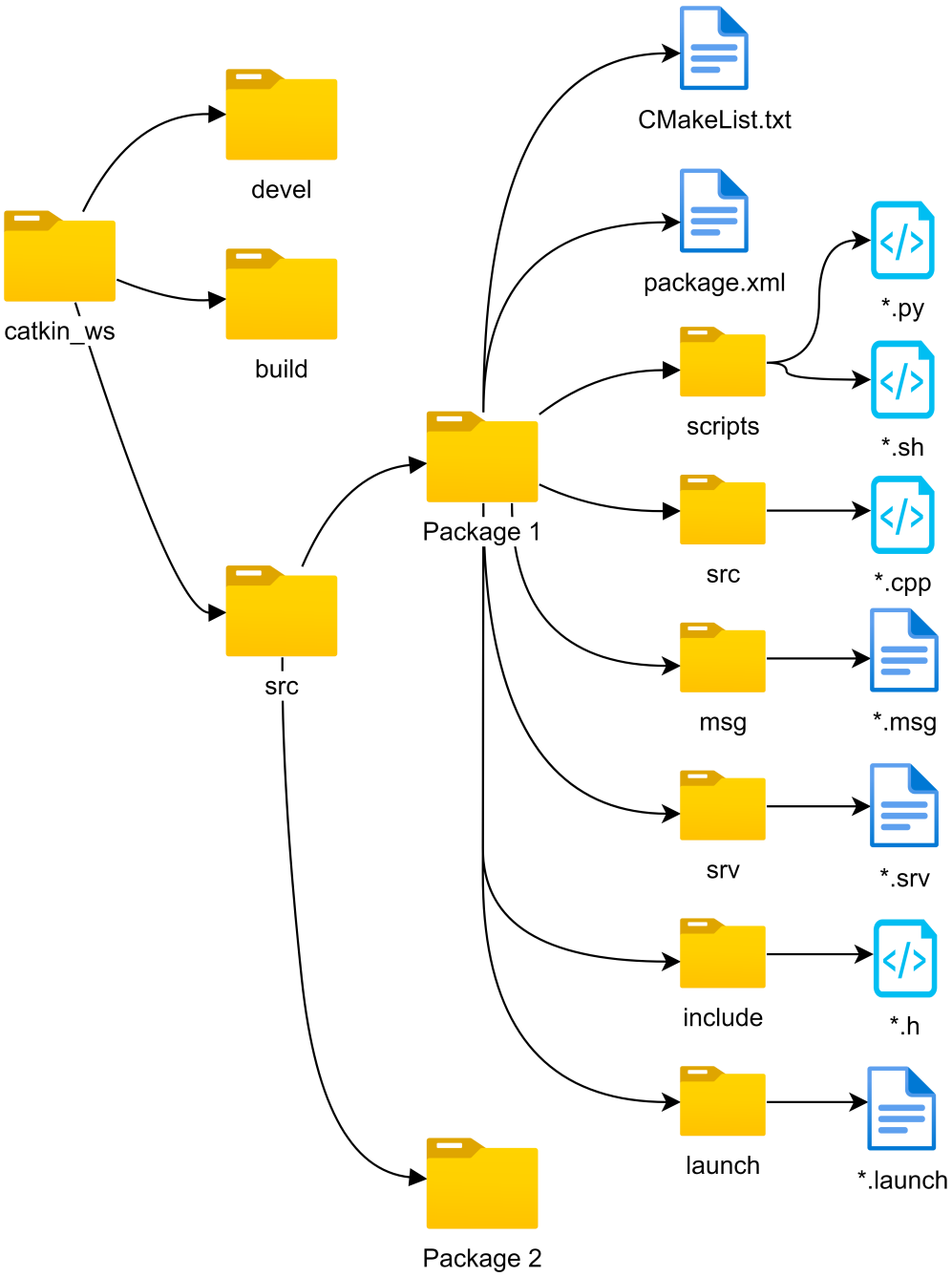


FIGURE 3.4: Files' structure for ROS workspace. [3]

`Catkin` serves as the ROS build system, comprising a set of tools utilized by ROS for generating executable programs, libraries, scripts, and interfaces that can be employed by other code. If you are developing your ROS code in C++, a good understanding of catkin is essential. However, since this book employs Python for its examples, we won't delve deeply into its intricacies. Nevertheless, we will explore its basic functionalities to some extent. For those interested in a more comprehensive understanding, the catkin wiki page is an excellent resource. If you are curious about why ROS has its dedicated build system, you can refer to the catkin conceptual overview wiki page. To install ros-melodic catkin workspace, you can use the following command:

Command

```
sudo apt-get install ros-melodic-catkin
```

ROS Catkin Workspace

The ros workspace has several folders as shown in Figure 3.4 above. Following, we will be looking at the function of each folder. [36]

src Forlder The 'src' directory within the catkin workspace serves as the designated location for creating or importing new packages from repositories. It's important to note that ROS packages are only built and turned into executables when they reside in the 'src' directory. When the 'catkin_make' command is executed from the workspace directory, it scans the 'src' folder, building each package found there.

build Folder When the 'catkin_make' command is executed within the ROS workspace, the catkin tool generates certain build files and intermediate CMake cache files within the 'build' directory. These cache files play a crucial role in preventing the need to rebuild all packages each time you run 'catkin_make.' For example, if you initially build five packages and subsequently introduce a new package to the 'src' folder, only the new package will be built during the next 'catkin_make' command. This efficiency is achieved through the utilization of cache files within the 'build' directory. It's important to note that deleting the 'build' folder will trigger a complete rebuild of all packages.

devel Folder When 'catkin_make' is executed, it triggers the build process for each package, resulting in the creation of target executables if the build is successful. These executables are saved within the 'devel' folder, which contains shell script files designed to incorporate the current workspace into the ROS workspace path. Access to the packages within the current workspace is only enabled when this script is executed. Typically, the following command is employed for this purpose.

Command

```
source ~/<workspace_name>/devel/setup.bash
```

3.4 Nodes in ROS

Definition

In the context of ROS, a node represents a computational process that undertakes specific tasks. These nodes collaborate within a graph structure and communicate through topics, RPC services, and the Parameter Server. [45]

The architecture encourages the deployment of numerous fine-grained nodes in a robot control system. For instance, individual nodes might be responsible for tasks like managing a laser range-finder, controlling wheel motors, handling localization, executing path planning, providing a graphical system view, and more.

3.4.1 Benefits of Using Nodes

1. **Fault Tolerance**: Nodes enhance fault tolerance by isolating crashes to individual components. If one node fails, it doesn't necessarily affect the entire system.
2. **Reduced Code Complexity**: Compared to monolithic systems, the use of nodes helps reduce code complexity. Each node focuses on specific functionalities, making the system more modular and easier to understand.
3. **Implementation Flexibility**: Nodes expose a minimal API to the overall system, allowing alternative implementations, even in different programming languages, to be seamlessly integrated.

3.4.2 Node Identification

Every running node in ROS possesses a unique graph resource name that distinguishes it within the system. For instance, "/hokuyo_node" might be the identifier for a Hokuyo driver responsible for broadcasting laser scans.

3.4.3 Node Type

Nodes also have a node type, simplifying the referencing of a node executable on the filesystem. Node types are determined by package resource names, incorporating both the node's package and the node executable file's name. When resolving a node type, ROS searches for all executables in the specified package and selects the first one it encounters. It's crucial to avoid producing different executables with the same name within the same package to prevent conflicts.

A ROS node is written with the use of a ROS client library, such as roscpp or rospy.

3.5 ROS Master Communication

The ROS Master functions as a provider of naming and registration services for the various nodes within the ROS system. It manages the tracking of publishers and subscribers to topics, as well as services. Essentially, the role of the Master is to facilitate the discovery of individual ROS nodes, allowing them to establish peer-to-peer communication.

Additionally, the Master offers the Parameter Server functionality.

To initiate the Master, the 'roscore' command is commonly used, which initiates the ROS Master alongside other indispensable components.

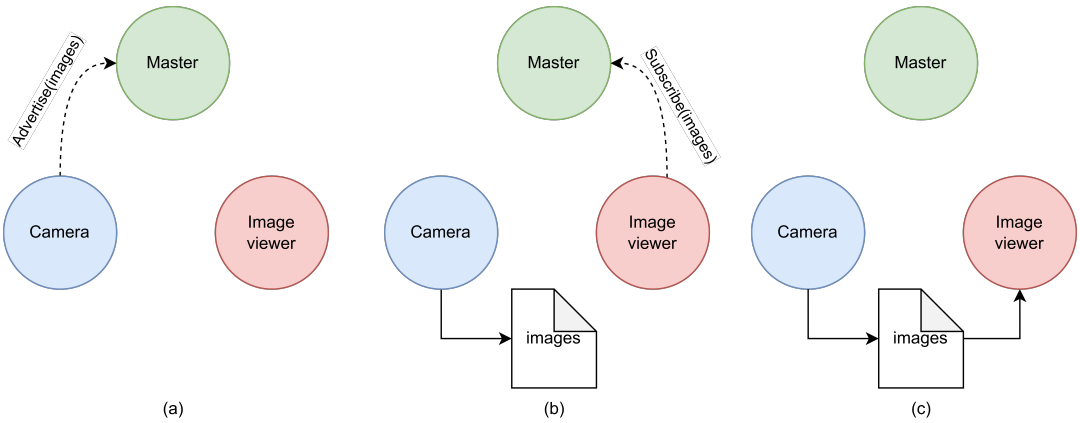


FIGURE 3.5: Master Name Service Example [77]

In the above-shown figure 3.5, let's consider a scenario involving two Nodes: a Camera node and an Image_viewer node. The typical sequence of events begins with the Camera node informing the master that it intends to publish images on the 'images' topic, as depicted in (a).

Subsequently, the Camera node commences publishing images to the 'images' topic. However, as there are no subscribers to this topic, no data is transmitted. In (b), the Image_viewer expresses its interest in subscribing to the 'images' topic to check if any images are available.

With both a publisher and a subscriber now present for the 'images' topic, the master node facilitates the awareness of Camera and Image_viewer to each other's existence, allowing them to commence the exchange of images, as illustrated in part (c) of the figure.

The communication models that are going to be described throughout the upcoming sections are the following:

1. Publisher-Subscriber
2. Server-Client Service
3. Server-Client Action

3.5.1 .bashrc File

It's essential to highlight the significance of the .bashrc file for comprehending the ROS-master communication setup. As discussed in the roscore section (3.3.2), several environment variables, such as ROS_MASTER_URI, are established during roscore initialization, often containing a URL in the format `http://hostname:11311/`, where the `hostname` represents the IP address of the ROS-master computer. The .bashrc, an abbreviation for **bash r**ead **c**ommand, serves as a configuration file for the Bash shell environment. Upon the initiation of an interactive Bash shell session, the .bashrc script file executes, encompassing diverse comments, configurations, **environment variable settings**, and functions aimed at customizing the shell experience and automating tasks. In order to open the .bashrc file in a text editor. For example, if you use nano, we run:

Command

```
nano ~/.bashrc
```

Alternatively, if you use Vim, run the following command:

Command

```
vim ~/.bashrc
```

Lastly, we edit the .bashrc file and add the following environmental variables using the export keyword. "10.10.10.10" is used in this example, because it's the IP address of the WiFi-communication in Niryo-Ned2. [21]

```
.
.
.
export ROS_MASTER_URI=http://10.10.10.10:11311/
export ROS_IP=10.10.10.10.102
```

3.6 Publishers-Subscribers

One fundamental aspect of communication in ROS involves the process of sending or publishing messages from a node to a designated topic, followed by the counterpart action of listening or subscribing to those messages from another node.

The intricacies of publishing and subscribing are pivotal elements in the ROS framework, necessitating a prior understanding of topics within the system.

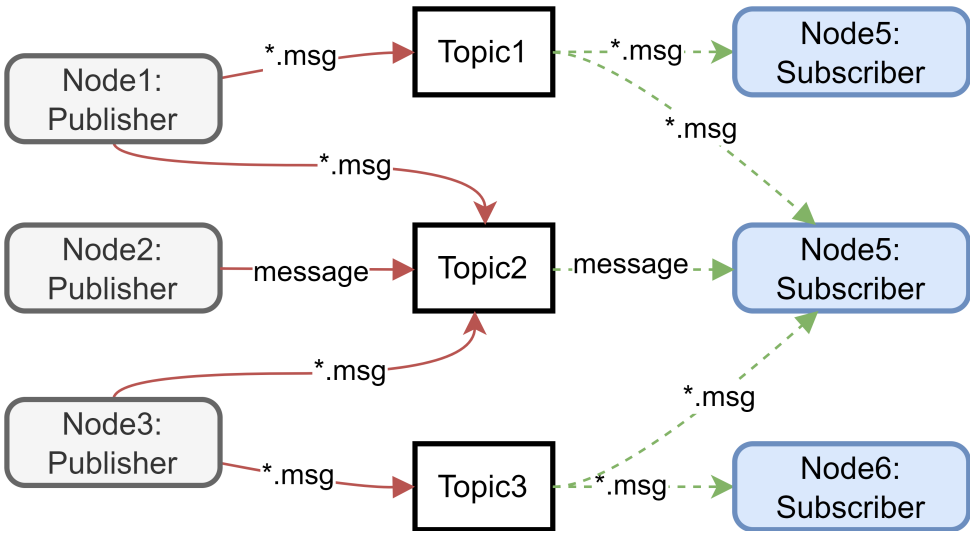


FIGURE 3.6: Publisher–Subscriber communication model.

3.6.1 Topics in ROS

Definition

In ROS, topics serve as named buses, facilitating the exchange of messages between nodes. They operate on anonymous publish/subscribe semantics, allowing the separation of information production from consumption. [20]

Nodes generally remain unaware of the entities they are communicating with. Instead, nodes interested in specific data subscribe to the relevant topic, while nodes generating data publish to that same topic. Multiple publishers and subscribers can exist for a single topic.

Key Characteristics:

1. **Unidirectional Streaming Communication**: Topics are designed for unidirectional, streaming communication. For remote procedure calls requiring a response, services are recommended, and the Parameter Server is used for maintaining small amounts of state.

2. **Topic Types**: Each topic is strongly typed by the ROS message type used for publishing. Nodes can only receive messages with a matching type, ensuring consistency in communication. The Master doesn't enforce type consistency among publishers, but subscribers won't establish transport unless the types match.

3. **Topic Transports**: ROS supports TCP/IP-based (TCPROS) and UDP-based (UDPROS) message transport. TCPROS is the default transport, streaming message data over persistent TCP/IP connections. UDPROS, currently supported only in roscpp, separates messages into UDP packets, making it suitable for low-latency, lossy tasks like teleoperation.

Nodes negotiate the desired transport at runtime, allowing flexibility and future additions.

Tools for Topics: The rostopic command-line tool is available for interacting with ROS topics. For instance, the command:

Command

```
$ rostopic list
```

provides a list of active topics. This tool is valuable for monitoring and managing the flow of information within the ROS system. Whereas, the command line:

Command

```
$ rostopic echo /topic_name
```

will display Messages published to /topic_name. Please refer to the rostopic page for more documentation.

3.6.2 Publishing a topic in ROS node

This section shows how to publish a topic in a ROS node. In Python, we use the following syntax with comments to explain each line. The upcoming code examples are taken from The Construct platform for robotics, please refer to their website or book for a more detailed explanation [71]:

```
#! /usr/bin/env python

# Import the Python library for ROS
import rospy
# Import the Int32 message from the std_msgs package
from std_msgs.msg import Int32

# Initiate a Node named 'topic_publisher'
rospy.init_node('topic_publisher')

# Create a Publisher object, that will publish on the /counter topic
# messages of type Int32
pub = rospy.Publisher('/counter', Int32, queue_size=1)

# Set a publish rate of 2 Hz
rate = rospy.Rate(2)
# Create a variable of type Int32
count = Int32()
# Initialize 'count' variable
count.data = 0

# Create a loop that will go until someone stops the program execution
while not rospy.is_shutdown():
  # Publish the message within the 'count' variable
  pub.publish(count)
  # Increment 'count' variable
  count.data += 1
  # Make sure the publish rate maintains at 2 Hz
```

```
29  rate.sleep()
```

LISTING 3.1: Publisher Node Example

In essence, this code initiates a ROS node and establishes a publisher responsible for continually transmitting a sequence of consecutive integers to the "/counter" topic. To encapsulate:

Definition

A publisher, in ROS terminology, denotes a node fulfilling the role of consistently sending messages to a specified topic. [20]

3.6.3 Messages in ROS

In ROS, communication between nodes is facilitated through topics, with messages serving as the carriers of information. Messages come in various types, and while users have the flexibility to create custom messages, leveraging the extensive collection of default ROS messages is often recommended for efficiency. [71]

Messages are formally defined in .msg files, typically housed within the msg directory of a ROS package. Understanding a message's structure and properties can be accomplished using the `rosmsg show <message>` command, offering insights into the composition of the message and its relevant details.

For example, let's try to get information about the std_msgs/Int32 message. Type the following command and check the output:

Command

$ rosmsg show std_msgs/Int32

Shell Output:

```
[std_msgs/Int32]:
int32 data
```

For instance, the Int32 message, commonly used in ROS, encapsulates a single variable labeled data with the data type int32. Originating from the std_msgs package, this message definition resides in the msg directory of the package. A closer inspection of the Int32.msg file can be conducted using the following command:

Command

$ roscd std_msgs/msg/

3.6.4 Subscribing to a topic in ROS node

As shown in Figure 3.6, the publisher nodes are continuously sending messages to a topic/s, and on the other hand, subscriber nodes are listening to a topic or more at a certain point in time.

let's explain a code inside a simple_topic_subscriber.py example script:

```
#! /usr/bin/env python

import rospy
from std_msgs.msg import Int32

# Define a function called 'callback' that receives a parameter
def callback(msg): # named 'msg'

  # Print the value 'data' inside the 'msg' parameter
  print (msg.data)

# Initiate a Node called 'topic_subscriber'
rospy.init_node('topic_subscriber')

# Create a Subscriber object that will listen to the /counter topic and
    will call the 'callback' function each time it reads something
   from the topic
sub = rospy.Subscriber('/counter', Int32, callback)

# Create a loop that will keep the program in execution
rospy.spin()
```

LISTING 3.2: Subscriber Node Example, ref: The Construct

In the described scenario, a subscriber node has been effectively implemented to monitor the `/counter` topic. When receiving data, the node triggers a designated function responsible for printing the message. Initially, the system remained inactive as there were no publishers transmitting data to `/counter topic`. However, upon executing the `rostopic pub` command, a message was successfully published to the /counter topic. Consequently, the associated function executed, printing the numeric value to the console. This event was also observable in the output of the `rostopic echo` command.

3.7 Services

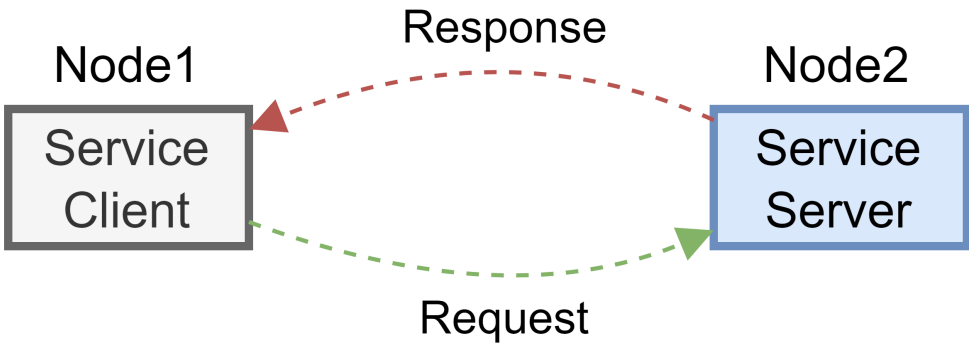


FIGURE 3.7: Server-Client Service communication

3.7.1 Service-Clients node in ROS

```python
#! /usr/bin/env python

import rospy
# Import the service message used by the service /trajectory_by_name
from trajectory_by_name_srv.srv import TrajByName, TrajByNameRequest
import sys

# Initialise a ROS node with the name service_client
rospy.init_node('service_client')
# Wait for the service client /trajectory_by_name to be running
rospy.wait_for_service('/trajectory_by_name')
# Create the connection to the service
traj_by_name_service = rospy.ServiceProxy('/trajectory_by_name',
    TrajByName)
# Create an object of type TrajByNameRequest
traj_by_name_object = TrajByNameRequest()
# Fill the variable traj_name of this object with the desired value
traj_by_name_object.traj_name = "release_food"
# Send through the connection the name of the trajectory to be executed
    by the robot
result = traj_by_name_service(traj_by_name_object)
# Print the result given by the service called
print(result)
```

LISTING 3.3: Simple Service-Clients node example, ref: The Construct

The provided Python script is designed to create a service client node in ROS that communicates with a service named '`/trajectory_by_name`'. Let's break down the script into sequential paragraphs to explain its functionality [71]:

1. **Shebang and Import Statements:** The script begins with the shebang line specifying the Python interpreter to be used. It then imports the necessary ROS packages, including '`rospy`' for interacting with ROS, and the service message classes ('`TrajByName`' and '`TrajByNameRequest`') generated from the '`trajectory_by_name_srv`' package.

2. **Initialize ROS Node:** The first line initializes a ROS node named '`service_client`'. A ROS node is a process that performs computation in the ROS environment.

3. **Wait for Service Availability:** The script waits for the service '/trajectory_by_name' to be available before proceeding. This ensures that the service server is up and running.
4. **Create Service Proxy:** A service proxy ('traj_by_name_service') is created to establish a connection to the '/trajectory_by_name service'. This proxy allows the script to call the service as if it were a local function.
5. **Create Service Request Object:** An object ('traj_by_name_object') of type 'TrajByNameRequest' is instantiated. This object is used to store the request data that will be sent to the service.
6. **Set Service Request Parameters:** The 'traj_name' attribute of the request object is set to the desired trajectory name, in this case, "release_food."
7. **Invoke the Service:** The script calls the service using the created service proxy ('traj_by_name_service') and passes the request object ('traj_by_name_object') as an argument. The result of the service call is stored in the 'result' variable.

In summary, this script initializes a ROS node, creates a service client, sends a service request to execute a trajectory named "release_food," and prints the result obtained from the service call.

3.7.2 *.srv Files

In ROS, the specifications for request and response message types in services are encapsulated in .srv files. These files adhere to a specific structure, ensuring clarity and consistency in the definition of ROS services. The following is an example of a .srv file structure:

```
int32 duration          # The time (in seconds)
---                  # Three dashes always between request/s and response.
bool success          # Did it achieve it?
```

3.7.3 Configuration for Custom Service Compilation

For the seamless compilation of custom ROS services, modifications need to be applied to two essential files within the package structure. These files, namely CMakeLists.txt and package.xml, should be edited in a manner consistent with the approach taken for topics. This ensures the proper integration and compilation of custom services in the ROS package. [34]

Modification of CMakeLists.txt

To enable the compilation of custom ROS services, four key functions within the CMakeLists.txt file need to be modified. These modifications facilitate the integration of service messages into the ROS package [70]. Below is a breakdown of the necessary changes:

1. **find_package():**

   - This function specifies all the packages required to compile messages of topics, services, and actions.
   - The listed packages are for obtaining paths and are not directly imported for compilation.
   - The same packages listed here should be included in the package.xml file under 'build_depend'.

```
find_package(catkin REQUIRED COMPONENTS
std_msgs
message_generation
)
```

2. **add_service_files():**
   This function enumerates all the service messages defined in the srv folder of the package.
   For example:

```
add_service_files(
FILES
MyCustomServiceMessage.srv
)
```

3. **generate_messages():**

   - In this function, packages needed for the compilation of service messages are imported.
   - Dependencies on standard message types, like std_msgs, should be specified.

```
generate_messages(
DEPENDENCIES
std_msgs
)
```

4. **catkin_package():**

   - Here, list all the packages that will be required by anyone executing something from your package.
   - All packages listed here must also be included in package.xml under '<exec_depend>'.

```
catkin_package(
CATKIN_DEPENDS rospy
)
```

Modification of package.xml

To configure the package.xml file for the compilation of custom ROS services, specific modifications are required. Follow these steps to incorporate the necessary changes:

1. Open the package.xml file and add the message_generation package as a build dependency. This ensures that the package is available for message compilation during the build process.

```
<build_depend>message_generation</build_depend>
```

2. Include the following lines to specify the runtime dependencies for message generation. These lines ensure that message_runtime is considered both for building and executing programs within the package.

```
<build_export_depend>message_runtime</build_export_depend>
<exec_depend>message_runtime</exec_depend>
```

   At the end of the modifications, your package.xml file should resemble the following:

```
<?xml version="1.0"?>
<package format="2">
<name>my_custom_srv_msg_pkg</name>
<version>0.0.0</version>
<description>my_custom_srv_msg_pkg package</description>

<maintainer email="user@todo.todo">user</maintainer>

<license>TODO</license>

<buildtool_depend>catkin</buildtool_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
<build_depend>message_generation</build_depend>
<build_export_depend>rospy</build_export_depend>
<exec_depend>rospy</exec_depend>
<build_export_depend>std_msgs</build_export_depend>
<exec_depend>std_msgs</exec_depend>
<build_export_depend>message_runtime</build_export_depend>
<exec_depend>message_runtime</exec_depend>

<export>
</export>
</package>
```

Once you're done, compile your package and source the newly generated messages:

Command

```
$ roscd;cd ..
$ catkin_make
$ source devel/setup.bash
```

3.7.4 Service-Server node in ROS

To implement a custom service server in a ROS package after creating the necessary .srv file, you need to create a Python program that defines the server's behavior. Follow these steps to create a service server using the example provided [70]:

1. Create a Python script, let's name it '`custom_service_server.py`'. Make sure it has the necessary shebang line (`#!/usr/bin/env python`) at the beginning.

2. Import the required ROS packages and the message classes generated from your custom service message ('**MyCustomServiceMessage**' and '**MyCustomServiceMessageResponse**').

3. Define a callback function ('**my_callback**') that will be executed when the service is called. The request message is passed as an argument, and you can access its fields (e.g., '**request.duration**'). Create a response message ('**my_response**') based on the received request. [33]

4. Initialize the ROS node, and create a service named '**/my_service**' with the specified callback function ('**my_callback**').

5. Keep the service open and responsive by calling '**rospy.spin()**'.

```python
#! /usr/bin/env python

import rospy
# you import the service message python classes generated from
    MyCustomServiceMessage.srv
from my_custom_srv_msg_pkg.srv import MyCustomServiceMessage,
    MyCustomServiceMessageResponse


def my_callback(request):
    print("Request Data==> duration="+str(request.duration))
    my_response = MyCustomServiceMessageResponse()
    if request.duration > 5.0:
        my_response.success = True
    else:
        my_response.success = False
    # the service Response class, in this case
    MyCustomServiceMessageResponse
    return  my_response

rospy.init_node('service_client')
# create the Service called my_service with the defined callback
my_service = rospy.Service('/my_service', MyCustomServiceMessage ,
    my_callback)
rospy.spin() # maintain the service open.
```

LISTING 3.4: Simple Service-Server node example, ref: The Construct

Now, when a client sends a request to the '**/my_service**' service, the '**my_callback**' function will be executed, and a response will be generated based on the request's duration.

To run your service server, execute the script:

Command

```
$ rosrun package_name custom_service_server.py
```

This server will be ready to handle service requests from clients. The provided example checks if the duration of the request is greater than 5.0 and sets the success field in the response accordingly. You can customize the '**my_callback**' function based on your specific service requirements. [29]

Here is another instance of a script written by me for a vibration motor operating with a service-server model that is designed for implementation during the later stages of a powder dosing procedure:

```
#!/usr/bin/env python

import time
import rospy

from fx_powder_dosage.srv import VibrationMotor, VibrationMotorRequest, VibrationMotorResponse
from niryo_robot_rpi.srv import SetDigitalIO, SetDigitalIORequest


def digital_out_do4(value=0):
    """Making a DO4 pin ON or OFF.

    param: value (int) -> off for 0, and on for any other value.
    """
    service_name = '/niryo_robot_rpi/set_digital_io'
    rospy.wait_for_service(service_name, 2)

    service_call = rospy.ServiceProxy(service_name, SetDigitalIO)
    request = SetDigitalIORequest()

    request.name = 'DO4'
    request.value = bool(value)

    response = service_call(request)

def callback_fun(req):
    on_off = req.on_off # 1 for ON, 0 for OFF
    freq = req.freq # The frequency [Hz]
    duty_cyc = req.duty_cycle # [%]
    time_on = req.time_on # 0 if you want the motor to run until you shut it off. [s]
    period = 1/freq

    start = time.time()
    end = 0

    if (time_on > 0) and (on_off == 1):
        # we need to make this happening for a certain number of second (time_on)!
        while ((end-start) <= time_on):
            digital_out_do4(1) # send a high singal to DO4
            time.sleep((duty_cyc/100)*period)
            digital_out_do4(0) # send a low singal to DO4
            time.sleep((1 - duty_cyc/100)*period)
            end = time.time()

        print("The duration of the vibration is: {}".format(end-start))

    elif (time_on == 0) and (on_off == 1):
        digital_out_do4(1)
    else:
        digital_out_do4(0)

    response = VibrationMotorResponse(bool(on_off))
    return response

def motor_server():
    rospy.init_node('vibration_server')
```

```
57      s = rospy.Service('vibration_motor', VibrationMotor, callback_fun)
58      print('Server Is called!')
59      rospy.spin()
60
61  if __name__ == "__main__":
62      motor_server()
```

LISTING 3.5: Vibration-Motor Service-Server

From the previous code, we can see that we had 4 requests in this service-server, and one response:

```
int8 on_off
int32 freq
int16 duty_cycle
int16 time_on
---
bool response
```

3.8 Actions

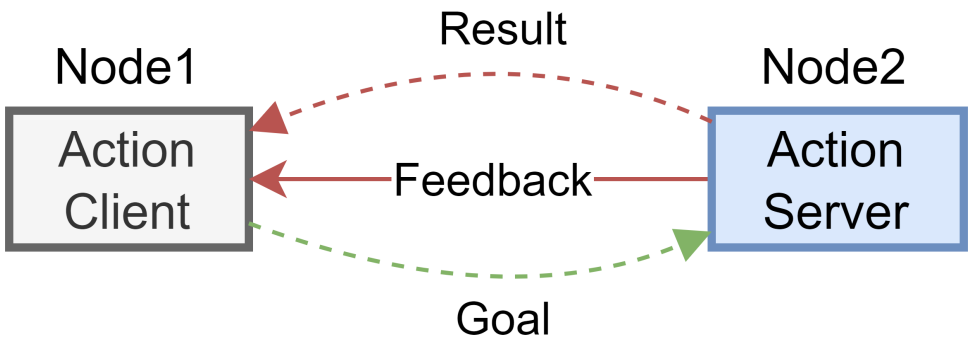


FIGURE 3.8: Server-Client Action communication

Actions represent the most intricate manner of communication in ROS. Leaning on Figure 3.8, an action client node kickstarts the communication by transmitting its destination message (goal) to an action server node. As the dissection at the core commences, the server continues relaying necessary feedback until the task at hand is achieved. Finally, after arriving at the desired goal, data reveals itself as a result back to the client completely its mission. This course builds and depends on the appropriate disposition of three parameters: the goal, feedback, and result message types. Consequently, creating a genuine connection of communications that highlights significant consequences. [24]

```
#goal
PoseStamped pose_goal
float64 wait_duration
- - -
#result
int32 status
string message
- - -
#feedback
int32 progression
```

In order for the client and server to interact, we must establish a few messages that they will use to communicate. This is done through an action specification, which outlines the Goal, Feedback, and Result messages that the client and server will exchange. [58]

- **Goal**: When performing actions, we utilize goals that are transmitted from an ActionClient to an ActionServer. To move the base, a PoseStamped message is sent to determine where the robot is to go. Managing the tilting laser scanner requires a goal specifying min and max angles, speed, and other settings.
- **Feedback**: For those who implement servers, feedback offers a method of informing an ActionClient of the gradual advancements of a goal. When moving the base, this could involve the current location of the robot along its route. Managing the angled laser scanner could mean providing information on the instance when the scanning process will conclude.
- **Result**: Once the goal is completed, the ActionServer sends a single result to the ActionClient, distinguishing it from feedback. This feature is extremely helpful for providing desired information needs. When working with a move base, the result's significance is minimal, serving to contain only the robot's final pose. Conversely, the tilting laser scanner's data-packed result could contain the point cloud generated from the requested scan.

3.9 ROS commands summary

There's a method where you can interact with ROS through the terminal window, enabling you to instruct commands into ROS and oversee or govern nodes in a given package using the command prompt. A digestible table 3.1 is available below to access a progression of available commands. [19]

Command	Action	Sample usage & illustration of some subcommands.
`roscore`	This starts the Master	$ `roscore`
`rosrun`	This runs an executable program and creates nodes	`$ rosrun [package name] [executable name]`
`rosnode`	This shows information about nodes and lists the active nodes	`$ rosnode info [node name]` `$ rosnode <subcommand>` **Subcommand**: `list`
`rostopic`	This shows information about ROS topics	`$ rostopic <subcommand> <topic name>` **Subcommands**: `echo, info, and type`
`rosmsg`	This shows information about the message types	`$ rosmsg <subcommand> [package name]/ [message type]` **Subcommands**: `show, type, and list`
`rosservice`	This displays the runtime information about various services and allows the display of messages being sent to a topic	`$ rosservice <subcommand> [service name]` **Subcommands**: `args, call, find, info, list, and type`
`rosparam`	This is used to get and set parameters (data) used by nodes	`$ rosparam <subcommand> [parameter]` **Subcommands**: `get, set, list, and delete`

TABLE 3.1: ROS commands summary with some examples context.

3.10 MoveIt! [65]

MoveIt![12] serves as the primary software framework within the Robot Operating System (ROS) for motion planning and mobile manipulation. It has garnered acclaim for its seamless integration with various robotic platforms, including the PR2 [78], Robonaut [2], and DARPA's Atlas robot. MoveIt! is primarily coded in C++, augmented by Python bindings to facilitate higher-level scripting. Embracing the fundamental principle of software reuse, advocated in the realm of robotics [42], MoveIt! adopts an agnostic approach towards robotic frameworks, such as ROS. This approach formally separates its core functionality and framework-specific elements, ensuring flexibility and adaptability, especially in inter-component communication.

By default, MoveIt! leverages the core ROS build and messaging systems. To facilitate effortless component swapping, MoveIt! extensively employs plugins across its functionality spectrum. This includes motion planning plugins (currently utilizing OMPL), collision detection (presently incorporating the Fast Collision Library (FCL) [47]), and kinematics plugins (employing the OROCOS Kinematics and Dynamics Library (KDL) [62] for both forward and inverse kinematics, accommodating generic arms alongside custom plugins).

MoveIt!'s principal application domain lies in manipulation, encompassing both stationary and mobile scenarios, across industrial, commercial, and

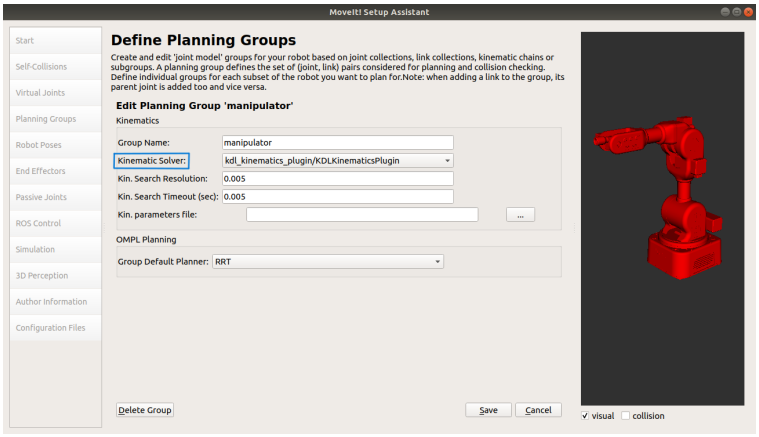


FIGURE 3.9: Choosing the kinematic solver for MoveIt planning group.

research settings. For a more comprehensive exploration of MoveIt!, interested readers are encouraged to refer to [12].

To plan and control the motion of your cobot through MoveIt, it is imperative to establish a MoveIt package within your Catkin workspace. The subsequent steps delineate the necessary procedures for the accurate creation of this package:

1. Launch the Moveit Setup Assistant.
2. Load a URDF or COLLADA robot model. For more details about URDF models, refer to section 2.5.2 in the previous chapter.
3. Generate a Self-Collision matrix to help the robot recognize which links could collide together in certain joint positions.
4. Create a planning group, which is crucial for choosing the kinematic solver for motion planning and control with MoveIt in the future. Refer to Figure 3.9.
5. Define some named pose positions.
6. Add a trajectory controller for each planning group, which is usually already defined in the URDF model.
7. Set the catkin workspace path, and finally, create the MoveIt package successfully.

Now, we can utilize the ROS-package '`moveit_commander`' Python interface to precisely plan the motion of the cobot. In this project, we employed '`rospy`', a pure Python client library for ROS, and '`moveit_commander`' to control the cobot and plan its motion.

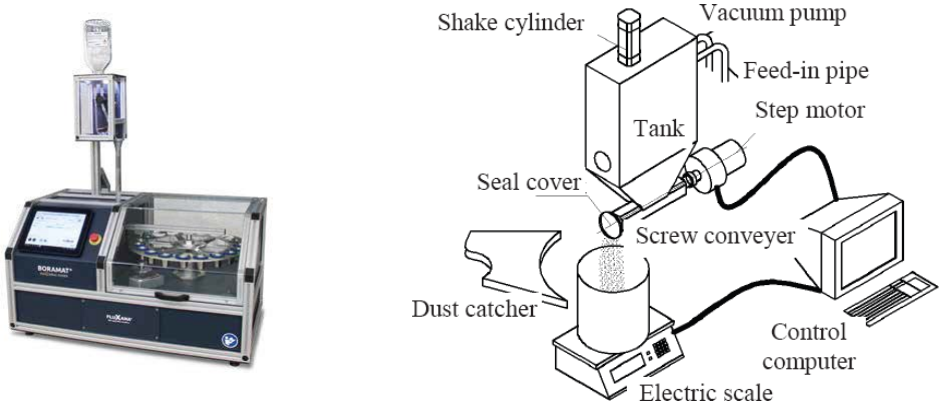
First, we need to define the planning group name to connect to using MoveIt as follows:

```
arm = moveit_commander.move_group.MoveGroupCommander("arm")
```

Next, we can utilize the MoveGroupCommander from MoveIt to plan the motion using joints, as demonstrated in appendix C.3, or set a pose goal, as shown in appendix C.4. Additionally, the forward kinematics ROS service from MoveIt can also be employed, as shown in appendix C.5.

Chapter 4

Basics of Powder Dosing



(a) FLUXANA's Boramat 18 [59]

(b) Hui Zhong and Zhongwen Ye's powder dosing machine [81]

FIGURE 4.1: Examples of available powder dosing machines

Applications for highly precise systems that dose and weigh granular or powdery substances are diverse and can be found in a variety of industries, such as electronic plants, medical production facilities, and packaging depots, to name just a few examples.

Various devices and controllers, such as FLUXANA's Boramat 18 [59] and Hui Zhong and Zhongwen Ye's highly accurate powder dosing machine developed in 2007 [81], have been invented to carry out powder dosage with precision. Nonetheless, our novel approach implements a cobot for the powder dosage process, providing remarkable adaptability with different systems and machines.

In this chapter, we will discuss the dependent and independent variables involved in our model approach and also explain a linear (polynomial) regression model used to approximate an equation for the collected data points from the experiments.

4.1 Dependant & Independent Variables

In academic research, variables refer to characteristics that exhibit various values, such as mass, frequency, angle, or speed. Researchers frequently engage with independent and dependent variables, manipulating or measuring them to explore cause-and-effect relationships [5].

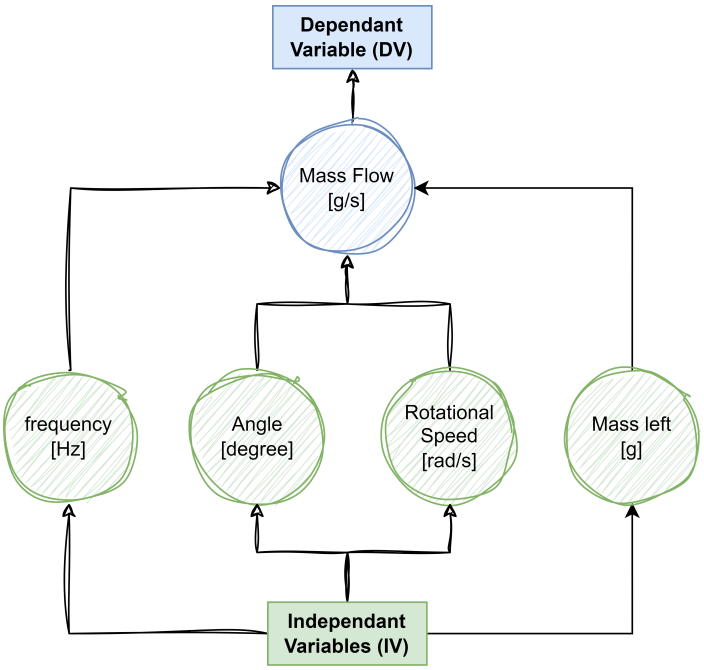


FIGURE 4.2: Dependant and Independent Variables in our research

Definition

An **independent variable** (IV) is the factor altered or controlled by the researcher in an experiment to examine its impact on the dependent variable.

It earns the label "independent" because its value remains uninfluenced by other study variables [5][26]. Also referred to as an explanatory or predictor variable, it plays a crucial role in experimental design [5].

Definition

Conversely, a **dependent variable** (DV) is the factor observed or measured in response to modifications in the independent variable.

Termed "dependent" because its outcome hinges on alterations made to the independent variable, it is also known as a response variable [5][26]. This variable serves as a key metric to gauge the impact of changes initiated in the independent variable [5].

In the course of our investigation, we are focused on tracking and documenting the *mass flow* of powder as the dependent variable. We are concurrently examining various independent factors, notably the vibration *frequency*, *angle*, *rotational speed*, and *mass left* within the crucible, as shown in Fig. 4.2.

To reduce the large number of possible combinations between the 4 independent variables, specific parameters are set specifically for each IV.

Mass left Starting only from 2 grams

Anlge α Four pre-defined α angles will be used, $5°, 13°, 16°, 21°$ degrees, as illustrated is Fig. 5.3.

Vibration Frequency 0Hz, 1Hz, and 10Hz.

Rotational Speed Restricted to 1.775, 1.0, 0.71 rad/s

In the next part, we will discuss our hypothesis for each independent variable.

4.1.1 Research Hypothesis

Definition

Hypothesis: an idea or explanation for something that is based on known facts but has not yet been proved. [according to the Cambridge Dictionary]

It is an educated guess that is based on prior knowledge and research [15].

The hypothesis of our independent variables with the dependent variable:

Frequency: Higher frequencies result in fewer powder droplets and of course reduced mass flow.

Angle: Increasing angles lead to greater mass flow, as depicted in Figure 5.3 where angle α is visible.

Rotation: Increased rotational speed results in higher mass flow.

Mass left: The mass left in the crucible is directly proportional to the powder mass flow.

Part II

Methodology, Experimental Set-up, and Results

Chapter 5

Methodology

In this chapter, we will elucidate the methodology employed in conducting the powder dosage experiments. It will not only encompass crucial Python code with MoveIt motion planning algorithms but also outline the comprehensive sequence logic governing the entire experiment. Additionally, we'll clarify significant functions utilized in constructing the sequence logic and algorithm before delving into the algorithm itself to ensure better comprehension.

5.1 Sequence Logic

In industrial automation, sequence logic refers to a meticulously arranged series of actions orchestrated to accomplish a particular outcome in a methodical and organized manner. These sequences serve as structured procedures executed by automated systems to achieve defined objectives. [16]

The sequential logic can be outlined in bullet points as follows:

1. **Start**: Calibration involves the automatic definition of the Zero positions for each axis of the robot.
2. Retrieve the Pre-saved trajectories.
3. Balance Setup.
4. Obtain the glass crucible.
5. Configure the dosing parameters.
6. Control the powder dosing process while receiving feedback from the balance.
7. Return the crucible upon completion of the dosing process.
8. Unset the balance, repositioning the metal crucible.
9. **End**: Return to the home position.

Each of these sequences will be explained in detail in the state diagram section 5.3.

5.2 Important Fuctions from FX_ROS Package

The FX_ROS.py package, developed by me, is dedicated to performing all the computational tasks related to motion planning using MoveIt and handling various ROS communication operations. This package efficiently manages operations such as subscribing to specific topics and initializing a ROS service client to invoke service servers for designated actions. It centralizes and streamlines these diverse functionalities within a single framework for enhanced convenience and effective ROS system interaction.

For more extensive details about the full script of FX_ROS package, please refer to the GitHub link in Appendix A.

Function	Description	Parameters
`Call_Aservice()`	Making a service client and sends a request to the server.	`service_name, type, request_name, req_args`
`Subscribe()`	Subscribe to a certain ros topic, and wait return the published message	`topic_name, type, msg_args`
`Get_joints()`	Subscribe to the '`/joint_states`' topic and return the current joint values	None
`get_pose()`	Subscribe to the '`/robot_state`' topic and return the current pose values	None
`move_to_joints()`	Move the cobot to a given joints values	`joints, arm_speed`
`Move_joint_axis()`	Move one joint axis at a time, either to a new position or shit the current position	`axis, new, add, arm_speed`
`Move_to_pose()`	Move the cobot to a given pose values	`pose_values, arm_speed`
`Move_pose_axis()`	Move one pose axis at a time, either to a new position or shit the current position	`axis, new, add, arm_speed`
`FK_Moveit()`	Make use of MoveIt!'s service-server to calculate and obtain the pose values based on the given joint values	`joints`

TABLE 5.1: Most important functions used in the research

This table showcases the key functions, listed in table 5.1. Each function name is clickable, leading directly to its function script. Alternatively, see Appendix C for comprehensive descriptions of each function. Worth mentioning, that the robot's movements are primarily planned and calculated by the functions in green, with a heavy reliance on joints over pose coordination. The intent was to obviate the need for any inverse kinematics calculations. Be that as it may, there are instances in which pose coordinations are still brought into play.

5.3 State Diagram

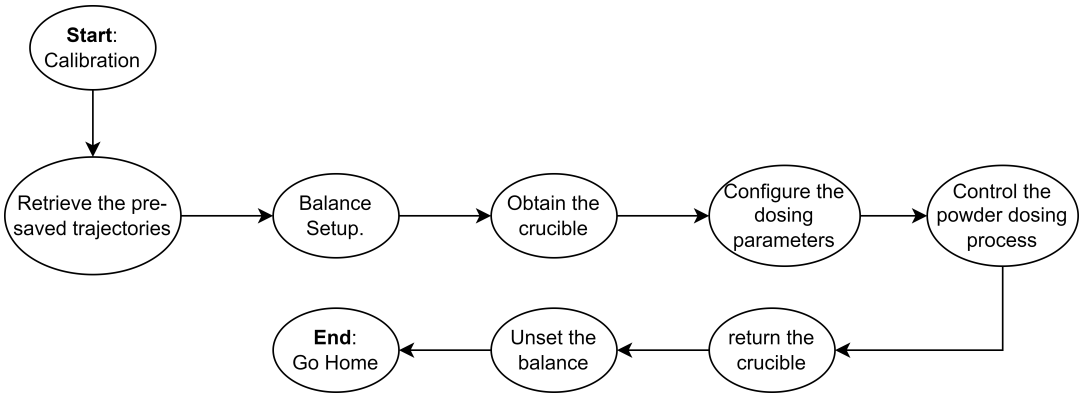


FIGURE 5.1: The sequence logic describing the full dosing process

The state diagram, resembling a state machine, comprises states, transitions, events, and activities. While the activity diagram elucidates the control flow between various activities across the involved objects, the state diagram demonstrates the control flow from one state to another within a singular object. It defines the progression of states an object experiences in reaction to events, including the corresponding responses to those events within the context of the sequence logic. [46]
In this section, we are going to understand more about each state in our sequence logic.

5.3.1 Zero State

In preparation for the experiments, we must first initialize the connection between ROS-master, represented by our Ned2 cobot, and the client, our Linux Ubuntu 18.04 OS laptop. To establish this connection, we take the following steps:

1. The acquisition of WiFi access to Ned2 [21] is necessary and can be established through the specifically assigned IP address of `10.10.10.10`.

2. Through SSH or a Secure Shell connection [79], connect with Ned2 through the Linux laptop, we initiate the connection by issuing the command:

Command

```
$ ssh niryo@10.10.10.10
$ niryo@10.10.10.10's password: robotics
```

3. Ensure that you modify the .bashrc file of both devices and assign the value of `http://10.10.10.10:11311/`to `ROS_MASTER_URI`, referring to chapter 3, section 3.5.1. Additionally, you should set two environmental variables for both devices in the following way:

```
export ROS_MASTER_URI=http://10.10.10.10:11311/
export ROS_IP=Device_IP
```

4. In order to shut down the `roscore` and reboot the launch nodes for recognition of the new setup of ros-master defined in the .bashrc file, one must eliminate all launch nodes. Follow the given three commands to achieve this:

Command

```
$ source ~/.bashrc
$ sudo service niryo_robot_ros stop
$ roslaunch niryo_robot_bringup niryo_ned2_robot.launch
```

Now the cobot is ready to calibrate and take commands.

5.3.2 Calibration

Ned2 undergoes an automatic calibration process, essential for its control, aligning the position of each motor with its respective controller [21]. The calibration sequence involves the following steps for the initial three axes:

1. The axis rotates in the direction depicted in the accompanying image until it halts.
2. It halts upon detecting the limit.
3. The axis retraces its motion in the opposite direction.
4. The axis rotates towards the limit stop.
5. It halts upon detecting the limit stop, establishing the Zero position at this point.

FIGURE 5.2: Calibration Process

To initiate the calibration of the motors, simply contact the service server at `'/niryo_robot/joints_interface/calibrate_motors'` and request a `value` of `1`. One way to request service is by using the terminal:

```
$ rosservice call /niryo_robot/joints_interface/calibrate_motors
[TAB] [TAB]
$ rosservice call /niryo_robot/joints_interface/calibrate_motors
"value: 1"
```

Alternatively, invoke the Python API to access the service via the GUI that we created.

```
def motor_cal():
    cal_service = '/niryo_robot/joints_interface/calibrate_motors'
    Call_Aservice(cal_service, type=SetInt, request_name=SetIntRequest,
     req_args={"value":1})
```

5.3.3 Retrieve the Pre-Saved Trajectories

In the following table 5.2, we will briefly describe the trajectories used in our motion planning algorithms to conduct the experiments in this research. Every time we open the GUI to conduct the experiment, we need to load a lookup table named "newbalance.xlsx" which contains all needed positions.

Position Name	Description	Number
pour(n)	Represents the placement of the metallic crucible, with n indicating the position, which can vary between 1 and 4.	4 positions.
pour(n)_out	The same as pour(n), whilst we move $8cm$ more in the z-axis. Check Fig. 5.4(b) for positions, and Fig. 5.5 for n indexing.	4 positions
balance & above_balance	The positions are needed to place the metallic crucible carefully on top of the balance.	2 positions
pick(n)	Refers to the position of the glass crucible.	4 positions
steady(n)	The position needed to approach the glass crucible from the bottom. Refer to Fig. 5.4(a)	4 positions
pick(n)_out1	1^{st} position needed to get the glass crucible out of its holder, without intersecting with the frame.	4 positions
pick(n)_out2	Takes the crucible away from the frame to avoid collisions when moving the given angle position.	4 positions
(α)angle	The position of TCP when dosing. Whereas α is representing the angle between the negative x-axis and the end-point of the cobot. Check Fig. 5.3	7 positions
(α)angle_rev	Same as the previous, however, with a different orientation of joint 6. Used specifically for 16°, and 21° for the algorithm to work.	2 positions

TABLE 5.2: The positions used to plan the trajectories.

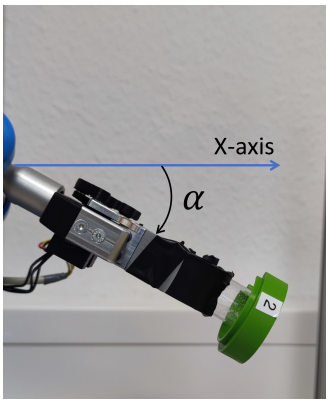
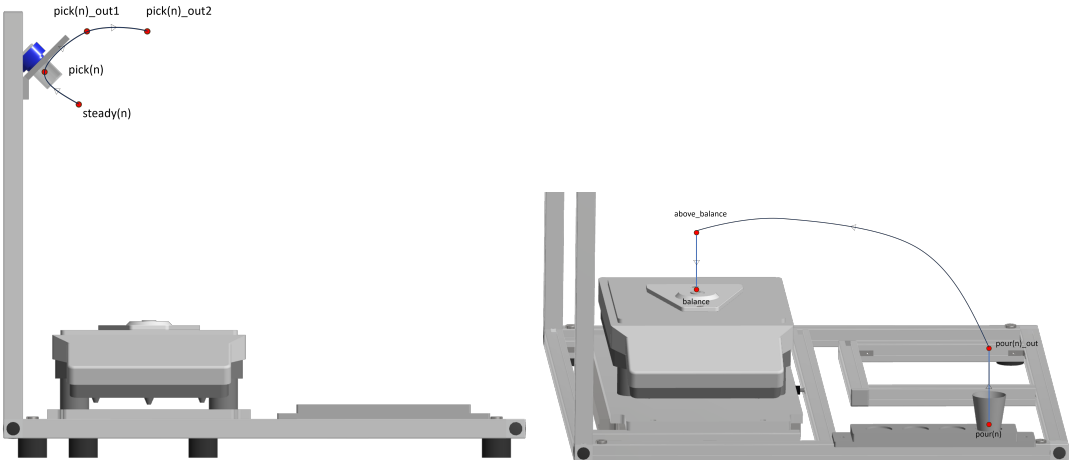
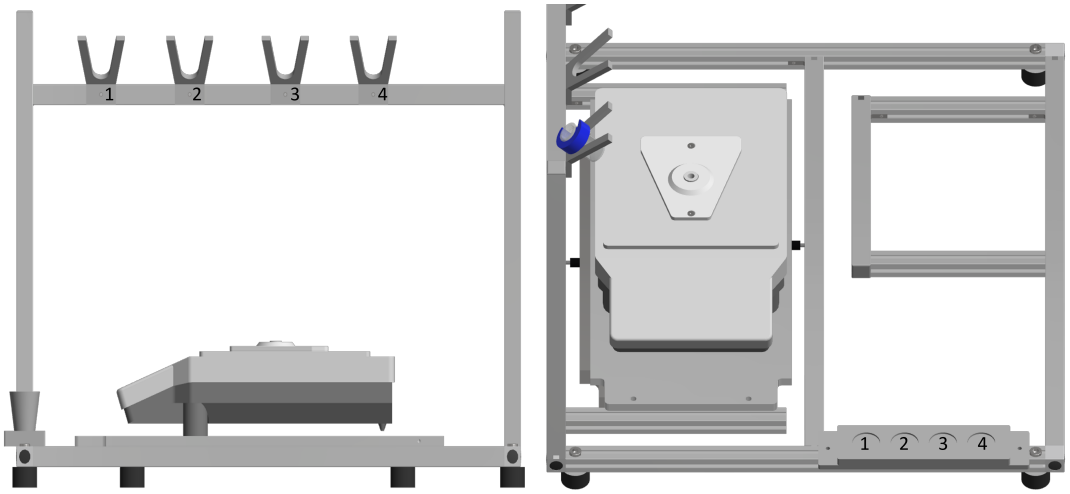


FIGURE 5.3: The angle α with reference to the $x - axis$.



(a) Pick & pour positions. (b) Balance Setup Positions.

FIGURE 5.4: Defining the pre-named positions used in the trajectory planning.



(a) Labeling of the glass crucibles. (b) Labeling of the metallic crucibles.

FIGURE 5.5: Indexing the crucibles' holders with numerical values.

5.3.4 Balance Set-Up

Throughout the explanation of the code in this section and the following sections, I will be using flowcharts as part of the UML (Unified Modeling Language). Flowcharts [10] serve as an effective means to conceptualize and depict the sequence of actions and logical flow within a program. These charts consist of interconnected geometric shapes linked by arrows, where each shape symbolizes a specific activity, and the arrows indicate the sequence of operations (referred to as flowlines). Typically, flowcharts are designed to flow from top to bottom and left to right. Particularly beneficial for novice programmers, flowcharts provide a visual representation of the program's structure, eliminating the need to concern oneself with language-specific syntax. Nevertheless, their practicality diminishes when dealing with extensive programs due to their cumbersome nature. [28] In the following Figure 5.6, you find the meaning of the used symbols in the flowchart.

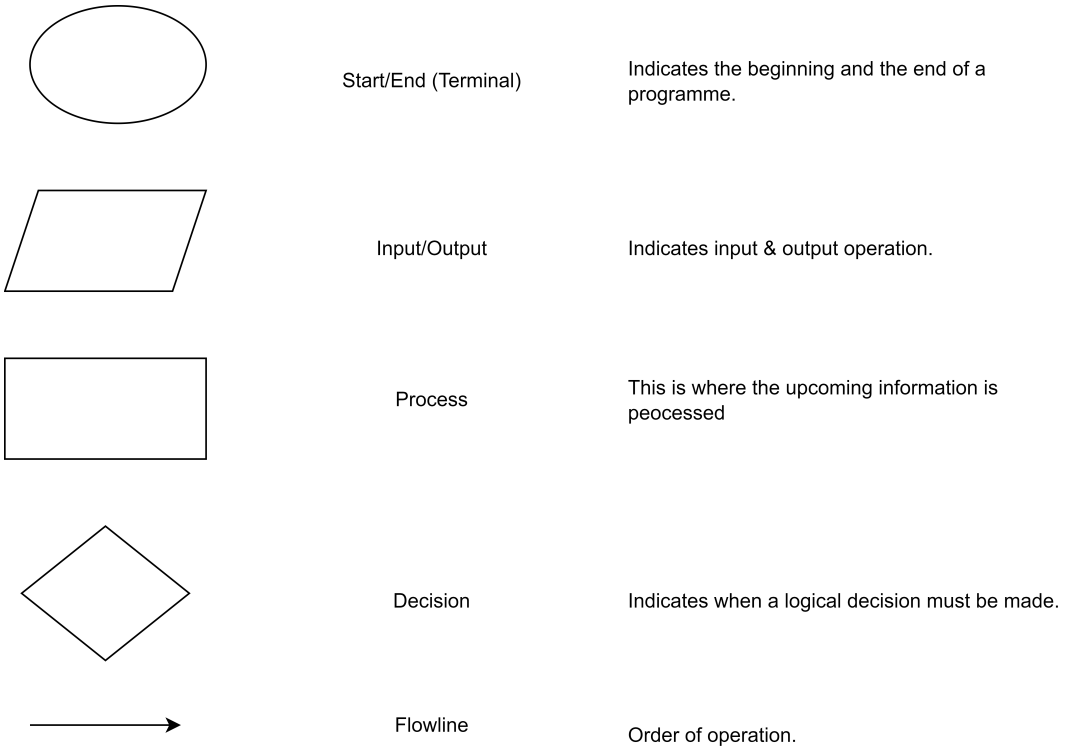


FIGURE 5.6: Common flowchart elements [28]

In this segment, we'll delve into the process of picking a metallic crucible, positioning it on the balance device, and preparing the balance for the dosing procedure. This step is crucial in ensuring the accuracy and precision of the measurements taken. So, let's explore the finer details of this essential task.

What are the variables that we have in this process?

Setup Global variable which represents whether the balance is set and which crucible also.

n An input that specifies which crucible we want to put on the balance.

Now, the process flowchart:

1. Start: Check whether Setup = True?
2. If yes, check whether the input n (the number of crucibles) is in the range from 1 to 4.
3. If yes, Load the needed positions: pour(n), pour(n)_out, balance, above_balance.
4. Open the gripper, and start moving by going to the 'straight forward' position
5. Wait for a second and go to pour_out (above the metallic crucible) and then move slowly towards the pour position.
6. Wait half a second and close the gripper.
7. After picking the crucible, get the arm up by moving back to the pour_out position.

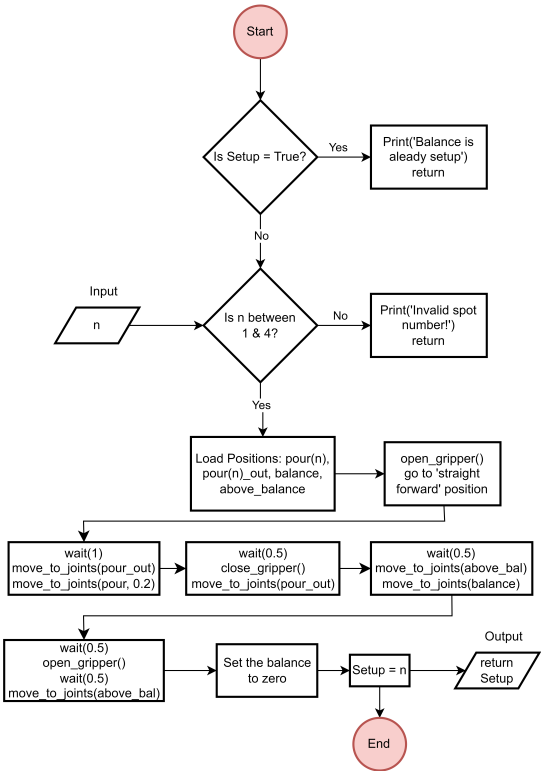


FIGURE 5.7: Balance Setup flowchart

8. Move now to the above_balance position, and slowly start to approach the balance to release the crucible.
9. Release the crucible by opening the gripper and return to the above_balance position.
10. Set the balance weight to Zero, and set the Setup global variable to be equal to n.
11. End: Return Setup.

5.3.5 Obtain the Crucible

Running up next, we'll explore the art and science of choosing a glass crucible and reaching a designated angle α, all while readying the balance for dosing. This method is key to maintaining pinpoint accuracy and finely honed measurement techniques.

What are the variables that we have in this process?

Setup Global variable described above.

Pour Global variable which represents whether the crucible is already picked and the index of the crucible as well.

n An input that specifies which crucible we want to pick for the dosing process.

α An input that specifies the angle as shown in Fig. 5.3.

Now, the process flowchart:

1. Start: Check whether the global variable Setup = True?
2. If yes, check whether the global variable Pour = True?
3. If no, Load the needed positions: steady(n), pick(n), pick(n)_out1, pick(n)_out2, (α)angle, 0angle, (α)angle_rev
4. Initiate the movement towards 'steady' location at 70% of maximum speed, gradually advancing towards the destination designated as 'pick' spot.
5. We advance toward pick_out1 and pick_out2 after sealing the gripper to keep the crucible beyond the holder's confines.
6. Move the end-effector away from the frame, and adjust the y-axis to shift 4cm in the negative direction.
7. Move to angle $\alpha = 0°$, then to α which is given as an input.
8. Set the Pour global variable to be equal to n.
9. End: Return Pour.

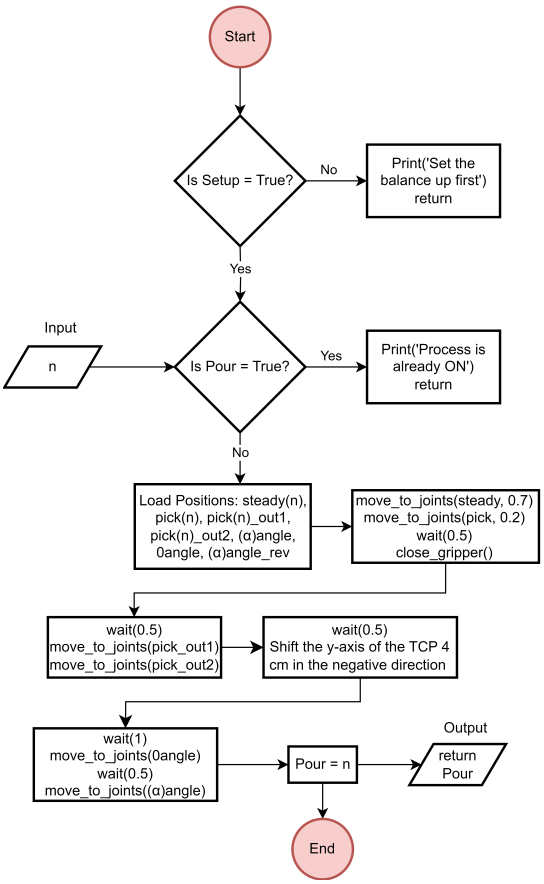


FIGURE 5.8: Obtain-Crucible flowchart

5.3.6 Do the Experiments

This section will offer a comprehensive overview of the various functions involved in the experiment.

What are the variables that we have in this process?

freq The frequency of vibration in Hz.

Speed Rotational speed in rad/s.

num_of_rot The amount of rotations that joint6 goes through before it transitions solely into vibration.

angle The angle α as shown in Fig. 5.3.

Now, the process flowchart:

1. Start: Setup Balance, as shown above in UML 5.7.
2. Obtain the crucible, as shown above in UML 5.8.
3. Begin conducting Trial 1 or Trial 2, and record measurements after each pulse until the intended dosage is obtained.
4. Return the crucible as shown in the below UML 5.10.
5. Unset the balance and return the metallic crucible, as shown in UML below 5.11.
6. End: Go to 'resting position'.

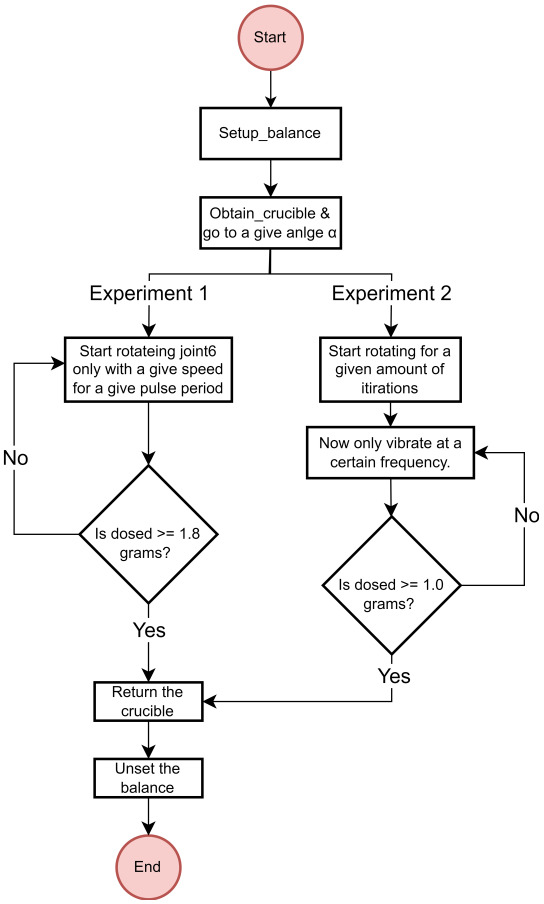


FIGURE 5.9: Do the Experiment flowchart

5.3.7 Return the Crucible

The section explains the process of putting the glass crucible back in its holder, outlining the necessary steps to follow.

What are the variables that we have in this process?

Pour Global variable which represents the index of the crucible in this case.

Now, the process flowchart:

1. Start: Check whether the global variable Pour = True?
2. If yes, Load the needed positions: steady(n), pick(n), pick(n)_out1, pick(n)_out2, 0angle.
   Knowing that n is known from the global variable 'Pour'.
3. Move only joint5 to 1.8 rads angle gradually, and start rotating joint6 to ensure that no more powder drops out.
4. We make our way towards pick_out2 and then onto pick_out1 to retrieve the crucible.
5. Begin your gentle approach towards the 'pick' position, then gradually loosen your grip to free the crucible.

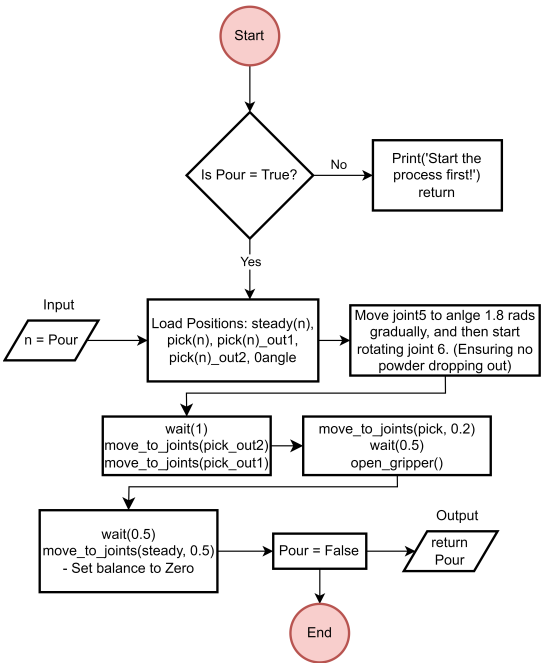


FIGURE 5.10: Return Crucible flowchart

6. Move back towards the 'steady' position and reset the balance back to zero.
7. Set the 'Pour' to False
8. End: Return Pour.

5.3.8 Unset the Balance

The subsequent paragraph illustrates the series of actions required to reset the metallic crucible to its designated container and complete the balance reset method.

What are the variables that we have in this process?

Setup This variable is global and pertains specifically to the crucible index.

Now, the process flowchart:

1. Start: Check whether the global variable Setup = True?
2. If yes, Load the needed positions: pour(n), pour(n)_out, balance, above_balance.
3. Open the gripper, and start moving by going to the 'straight forward' position
4. Move now to the above_balance position, and slowly start to approach the balance to pick the crucible.
5. Wait half a second and close the gripper.
6. Move back towards the 'above_balance' position.

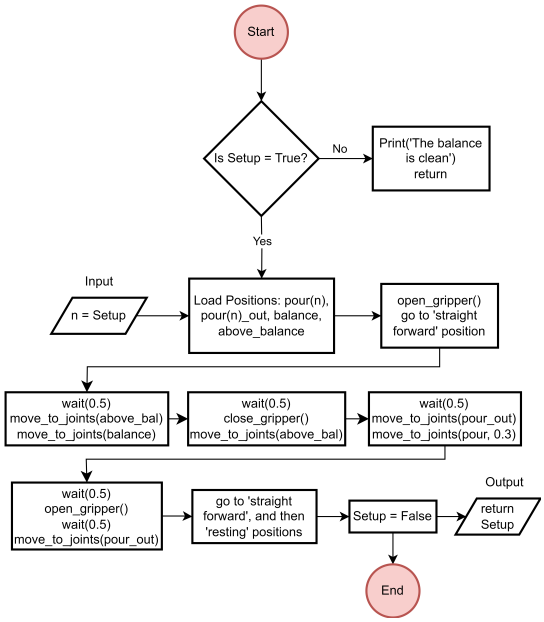


FIGURE 5.11: Unset balance flowchart

7. Wait for half a second and go to 'pour_out' (above the n crucible position) and then move slowly towards the 'pour' position.
8. Release the crucible by opening the gripper and return to the 'pour_out' position.
9. Go to 'straight forward' and then 'resting' position.
10. Set the 'Setup' to False.
11. End: Return Setup.

Chapter 6

Experimental Set-up

This chapter initiates a comprehensive exploration of the experimental setup. It commences with an overview of the workspace frame, encompassing the array of interconnected devices. These include various crucibles, the balance device, both its hardware and software components, the Ned2-cobot with its hardware configurations and software tools, precision validation procedures and concludes with an examination of the vibration motor, which is an auxiliary component integrated with the cobot.

6.1 Frame

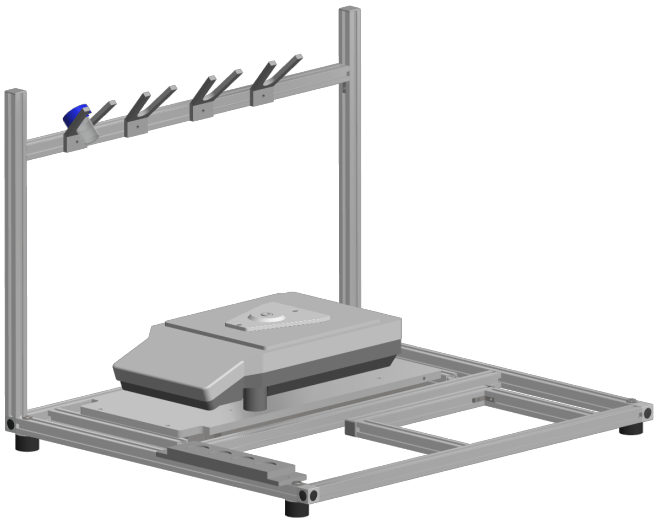
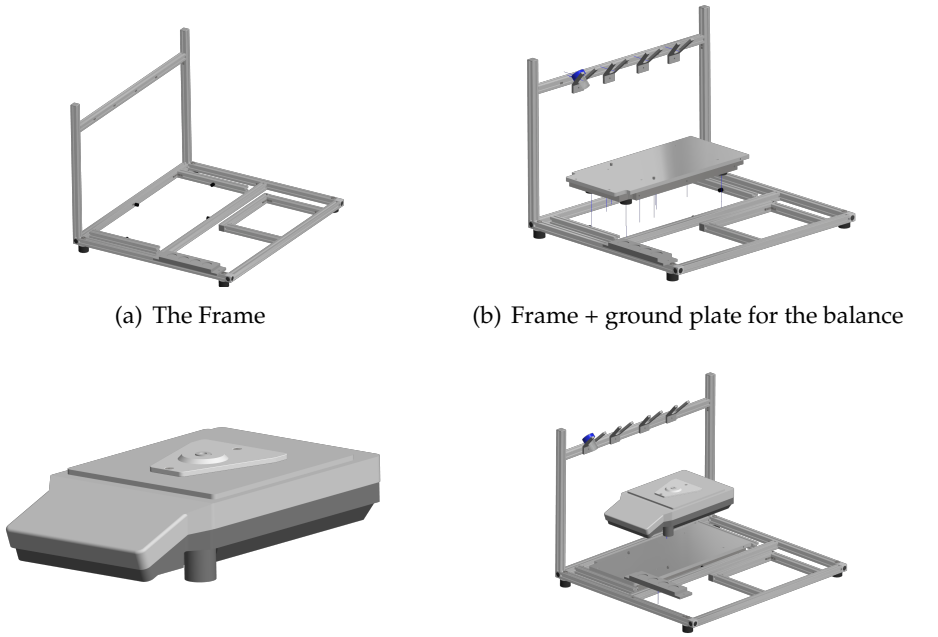


FIGURE 6.1: The Full Framework of System

In the realm of robotics, establishing a meticulously structured workspace framework is of paramount significance. This framework serves as the foundation encompassing all the requisite components with which the cobot will interact. As illustrated in Figure 6.1, this framework has been thoughtfully designed to meet these specifications.



(a) The Frame (b) Frame + ground plate for the balance

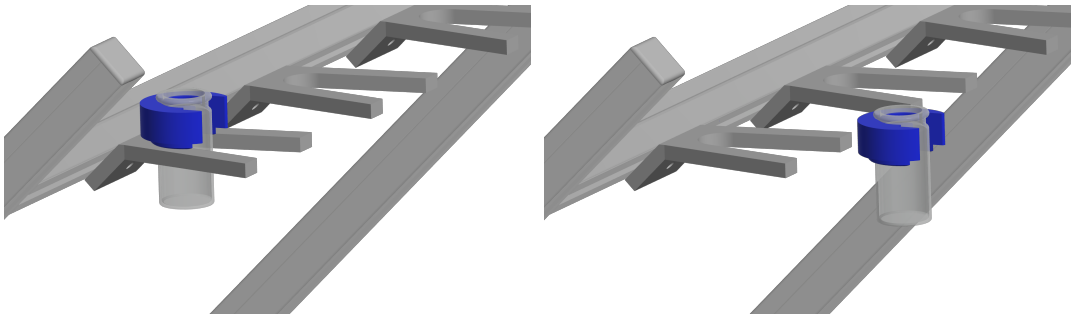
(c) The METTLER TOLEDO Balance [73] (d) With the balance and metal crucible hanger

FIGURE 6.2: Demonstration of the parts of the framework.

The framework encompasses various crucial components, as detailed below:

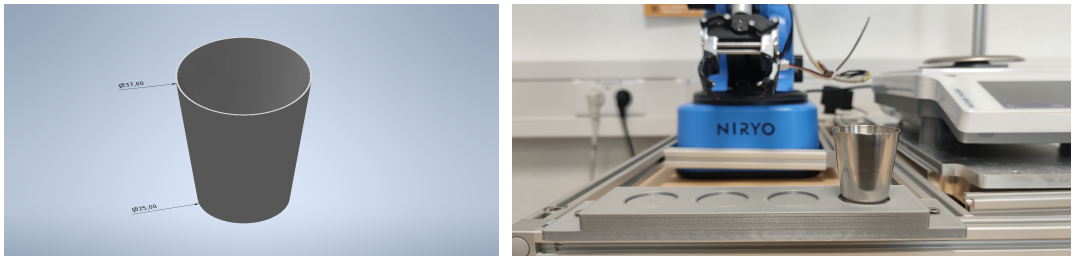
1. **The Frame** itself, depicted in part (a) of Figure 6.2. More comprehensive technical documentation of the frame's constituent parts and dimensions can be found in Appendix F.
2. **The Ground Plate of the Balance**, featuring a rubber base, is illustrated in part (b) of Figure 6.2. This element plays a pivotal role in ensuring the stability of the balance. Given its high sensitivity and precision, it is imperative to eliminate any potential transmission of motion or vibration
3. The Mettler Toldedo Me Balace, showcased in part (c) of Figure 6.2. For an in-depth understanding of the balance's specifications pertinent to our project, please refer to section 6.3.
4. **Four Glass Crucibles** and their corresponding hangers, documented comprehensively in section 6.2.
5. **Four Metal Crucibles** along with their hangers, as outlined in section 6.2.
6. The **Niryo Ned2 Cobot**, expounded upon in detail in section 6.4.
7. **Standing Rubber Buffers**, strategically incorporated to enhance the frame's stability.

Subsequent sections will delve into the intricate particulars and documentation of these essential components for our project.



(a) (b)

FIGURE 6.3: The glass crucible with its hangers.



(a) The technical design. (b) An actual crucible on the hanger.

FIGURE 6.4: The metal crucible and its hanger.

6.2 Crucibles

The crucibles play a central role in our project by serving as containers for the powder to be transferred from one crucible to another.

We will utilize two distinct types of crucibles:

- **Glass Crucibles**: These crucibles will predominantly be employed for powder dosing purposes.

  In the frame, we have incorporated four hangers for the glass crucibles, which can be observed in Figure 6.1. Additionally, the range of motion for the glass crucible is depicted in Figure 6.3. The crucible has an outer diameter of $22mm$, and an inner diameter of about $17mm$. More comprehensive information regarding the dimensions and technical specifications of the glass crucible can be found in Appendix E.

- **Metal Crucibles**: These crucibles will hold the powder after it has been dosed from the glass crucibles and will be utilized to measure the weight on the balance.

  The hanger serves the purpose of accommodating four crucibles, and you can observe its design in Figure 6.4. It possesses an upper diameter of 37mm and a lower diameter of 25mm, as visualized in Figure 6.4(a). For a more in-depth exploration of the technical design and drawings of this component, please consult the details provided in Appendix E, where comprehensive information is available.

6.3 Balance Device

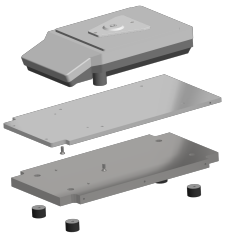


FIGURE 6.5: The balance with its metal plates and rubber buffers.

The precision balance employed in this study is a product of METTLER TOLEDO, a reputable German company. It offers an impressive readability of 0.1mg and a substantial weight capacity, approximately 220g, as referenced in [73]. For comprehensive insights into the specific components and technical specifications of this balance, please consult the information provided in Appendix D. The balance's operation, illustrated in Figure 6.5, involves its integration with two robust metal ground plates and the inclusion of four standing rubber buffers, which collectively contribute to the system's ensured stability during the weighing process.

6.4 Ned2 Collaborative Robot

Ned2 is a collaborative robot, often referred to as a cobot, developed by the French company Niryo [22]. This particular cobot has been purpose-built for educational and research applications, serving as a valuable tool for the development of proof of concepts and experimental work. In the context of this research, the Ned2 cobot played a pivotal role in conducting experiments.

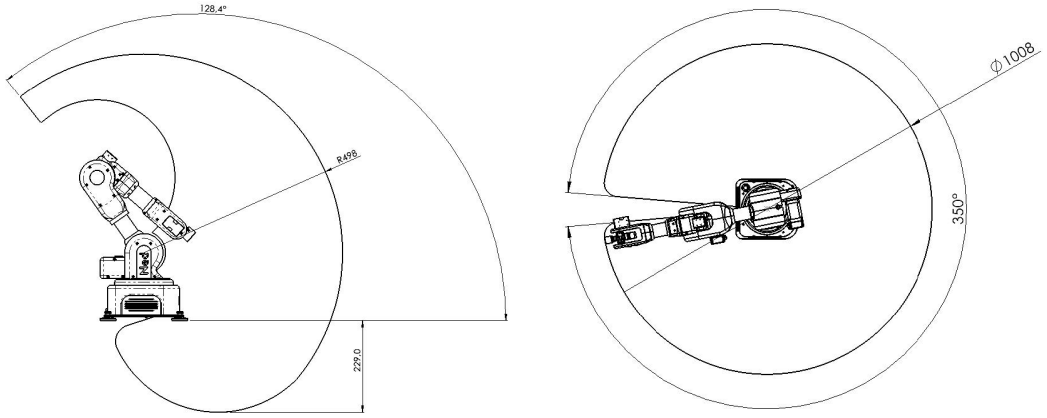
The forthcoming sections delve into an in-depth examination of the hardware specifications and software options offered by the Ned2 cobot.

FIGURE 6.6: Ned2 cobot provided by Niryo [22].

6.4.1 Hardware Configurations

Ned2 is a six-axis collaborative robot, based on open-source technologies. It is intended for education, research and Industry 4.0." [21]

Incorporating the same aluminum framework as its predecessor, Ned2 maintains its commitment to meeting your exacting standards in terms of durability, precision, and repeatability (with an accuracy, and a repeatability of 0.5 mm).



(a) (b)

FIGURE 6.8: The detailed workspace of Ned2 cobot.

Ned2 operates on the Ubuntu 18.04 platform and utilizes the ROS Melodic framework, capitalizing on the capabilities of the **Raspberry Pi 4**. This high-performance **64-bit ARM V8 processor**, coupled with **4GB of RAM**, empowers Ned2 to deliver enhanced performance.

(a) 2x XL430 (b) 1x XL330

FIGURE 6.7: The Servo motors used in Ned2.

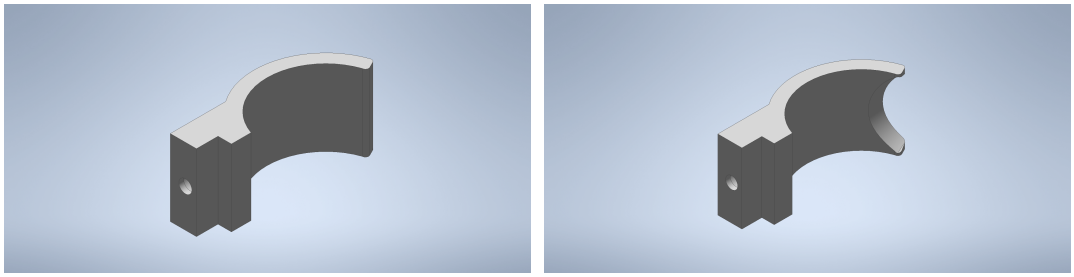
This iteration of Ned2 introduces advanced servo motors equipped with Silent Stepper Technology, significantly reducing the operational noise of the robot. As a six-axis robot, the device comprises a total of six distinct motors. The initial three motors are specifically customized and operate as silent Stepper motors, delivering 65 $N-cm$ stall torque. These are equipped with a custom-made control card developed by Niryo. The remaining three motors are servos, with two being XL430 type motors delivering 140 $N-cm$ stall torque, and one XL330 type motor providing 65 $N-cm$ stall torque, as shown in Figure 6.7 [21]. The technical specifications of Ned2 are described as shown in table 6.1 below.

Furthermore, each robot must operate within a well-defined workspace that allows for unrestricted movement. In our case, the workspace of Ned2, as depicted in Figure 5.4 (a) and (b), has been meticulously documented in accordance with Niryo company specifications [21]. Our frame was meticulously designed and engineered to align with the defined workspace of the cobot.

Parameters	**Value**
Weight (kg)	7
Payload (g)	300
Reach (mm)	440
Degree of freedom	6 rotating joints
Joints range (rad)	$2,949 \leq Joint1 \leq 2,949$ $-2,09 \leq Joint2 \leq 0,61$ $-1.34 \leq Joint3 \leq 1,57$ $-2,089 \leq Joint4 \leq 2,089$ $-1,919 \leq Joint5 \leq 1.922$ $-2,53 \leq Joint6 \leq -2,53$
Joints speed limit (rad/s)	$Joint1 \leq 0.785$ $Joint2 \leq 0.5235$ $Joint3 \leq 0.785$ $Joint4 \leq 1.57$ $Joint5 \leq 1.57$ $Joint6 \leq 1.775$
TCP max speed (mm/s)	468
Repeatability (mm)	+/- 0,5
Footprint (mm)	200x200
Power supply	Input: AC100-240V / 50-60Hz, 2,5A Output: DC 12V - 7A ; 5V - 7A
I/O power supply	5V
Inputs/Outputs Control panel	Digital input x1 Digital output x1
Robot interface	USB2.0 x2 USB3.0 x2 ETHERNET GIGABIT x1
Communication	Modbus TCP (master) TCP/IP
Materials	Aluminum ABS-PC (injection moulding)
Collision detection	Accelerometer & gyroscope in the control panel
Certification	CE Conformity

TABLE 6.1: Technical Specification of Ned2 [21]

Cobot's Gripper

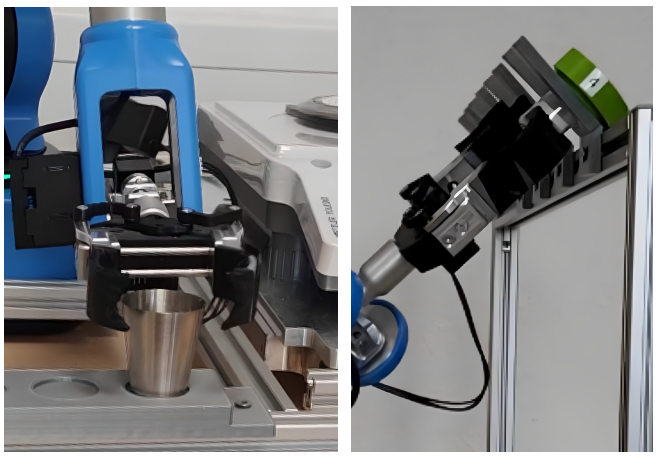


(a) The Initial phase of developing a gripper for the metal crucible only.

(b) The final phase of developing the gripper for both crucibles used in the research.

FIGURE 6.9: The development phase of the gripper.

While it's common to use general-purpose grippers such as those available from Niryo company, our research and development application necessitated a customized gripper. We have engineered a specialized gripper tailored to the unique demands of our project. In Figure 6.9, we can see the development phase of the gripper in order to fit the application for both glass and metal crucibles. The main tasks that the gripper is going to be used are holding the metal crucible from the middle, as shown in part (a) Figure 6.10, and holding the glass crucible from the bottom by the gripper's tip, as shown in part (b) Figure 6.10



(a) The initial task.

(b) The second task.

FIGURE 6.10: The main tasks of the gripper.

6.4.2 Software Tools

Ned2 represents a collaborative robot, hinging on the Ubuntu 18.04 platform and ROS Melodic—a widely adopted open-source solution in the field of robotics. Leveraging ROS, Ned2 offers an extensive array of libraries that empower users to create a wide spectrum of programs, from the simplest to the most intricate, thus ensuring adaptability to diverse operational requirements [21].

FIGURE 6.11: Software tools supported by Niryo in Ned2. [67]

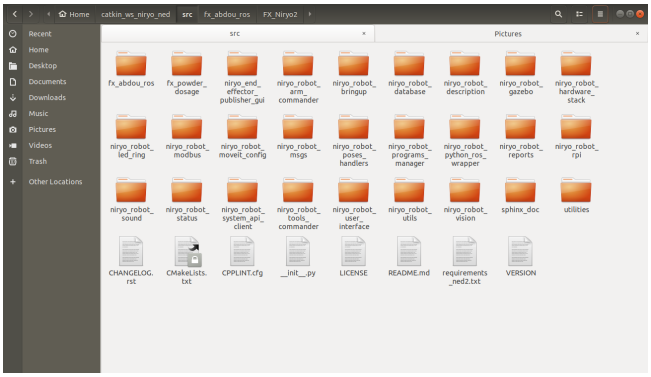


FIGURE 6.12: The Catkin workspace in Ned2

The software tools available for connecting and controlling the Niryo Ned2 cobot are extensive and diverse. They primarily include Niryo Studio, an interface offering multiple connectivity options such as hotspot, Wi-Fi, and Ethernet modes, each prompting calibration. This interface allows users to configure workspaces, adjust camera parameters, and control the conveyor belt. The cobot operates on ROS, featuring a range of proprietary and ready-made packages available on GitHub. It integrates motion planning capabilities using the MoveIt package, particularly employing the KDL kinematics plugin [62]. Thanks to Tensorflow (open-source machine learning tool developed by Google) [69], and OpenCV (open-source computer vision and machine learning software library) [68] training a computer vision-based model and recognizing objects is possible with Ned2. It supports multiple programming languages, including Blockly, Python, MATLAB [9], and interfaces with Arduino controllers [66], Modbus TCP servers, and various other languages via a TCP/IP server. Moreover, it provides the convenience of running Python scripts directly on the robot using the Python ROS wrapper (PyNiryo), making code execution simpler. Extensive tutorials, online documentation [21], and a dedicated support team is available for in-depth exploration and assistance. [23]
However, This research is conducted using the ROS-Melodic version, and communication is done through ROS-master communication, with mainly Python scripts.

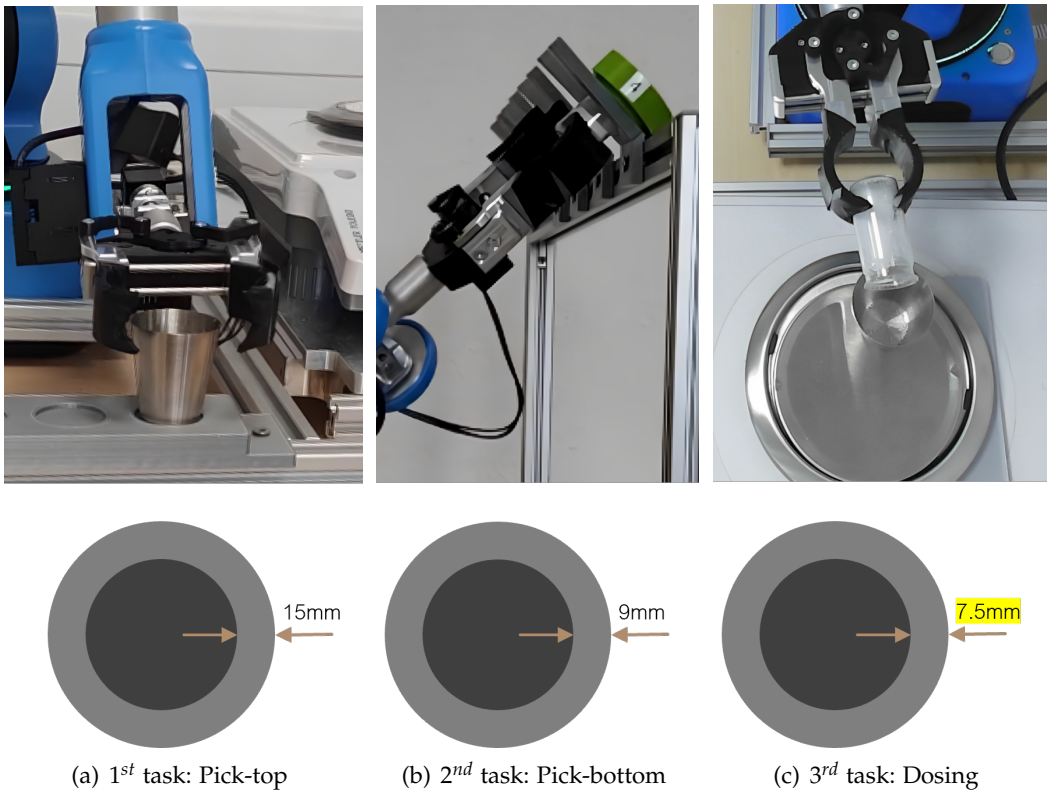
ROS-Melodic

Since the ROS Melodic version is installed and used in the raspberry in Ned2 cobot, we had to use Ubuntu 18.4 on the PC where we connected with the robot through ROS-Master communication. Please refer to section 3.1 for more info about Linux.
Niryo has provided various ROS packages in their workspace, as shown in Figure 6.12 below.

6.5 Precision of Ned2

In this section, we will assess the maximum precision achievable by the Ned2 cobot and determine if it meets the requirements for the experiment's tasks.

6.5.1 Required Precision



(a) 1^{st} task: Pick-top (b) 2^{nd} task: Pick-bottom (c) 3^{rd} task: Dosing

FIGURE 6.13: The required precision for each task in our research

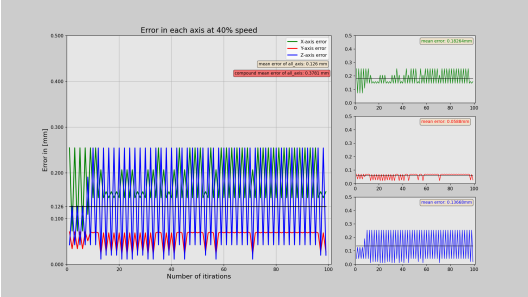
As depicted in Figure 6.13 above, the dosing process involves three tasks that are illustrated by the difference in radius between the cobot's gripper and the crucibles. It is important to note that the values presented in the figure assume that the crucibles are always approached from their center.

- In the third task, it is evident that our minimum precision requirement is 7.5 mm.
- We have determined that a minimum precision requirement of $\frac{7.5mm}{2} = 3mm$ is necessary. This is because our previous assumption of approaching the crucibles at the center is not always valid.

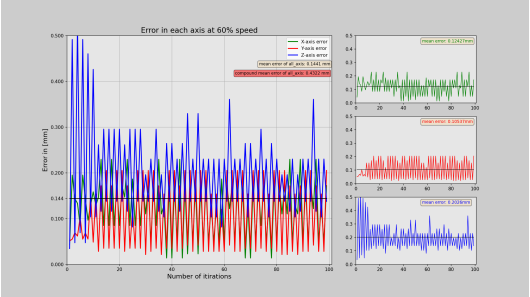
6.5.2 Precision Validation

To validate the precision of the Ned2 cobot, I performed the following steps:

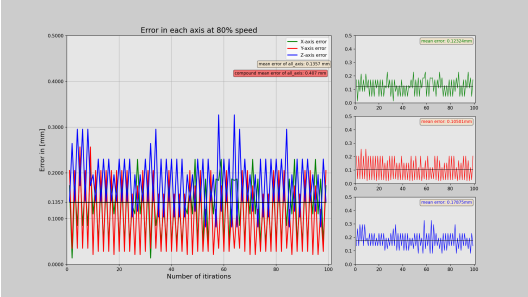
1. I moved the cobot arm to two different positions repeatedly for 100 iterations.



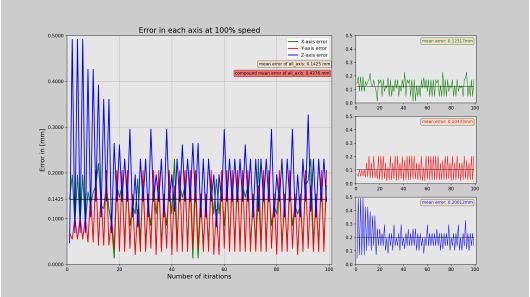
(a) At 40% of max. velocity



(b) At 60% of max. velocity



(c) At 80% of max. velocity



(d) At 100% of max. velocity

FIGURE 6.14: Recorded errors were measured along the x, y, and z axes for various velocities.

2. I retrieved the pose values from the "`RobotState`" topic in ROS, which are obtained from the sensor readings.

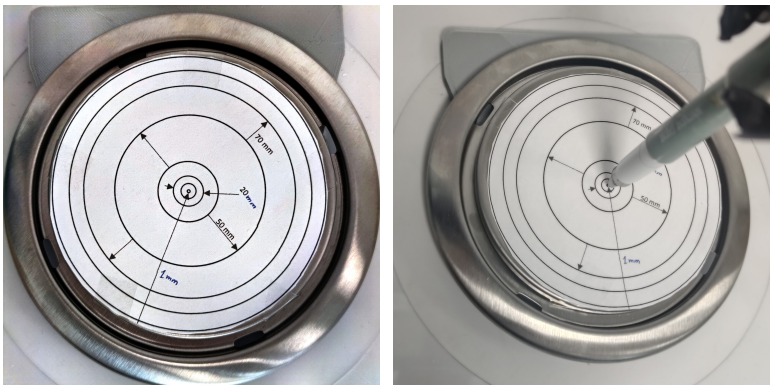
3. I checked the error in the x, y, and z axes of the TCP (Tool Center Point).

4. Repeat the process for various velocities and ensure that the error remains consistent. Check the results in Figure 6.14.

5. In order to ensure the accuracy of the graph results, it is crucial to visually represent the process and ensure that the data is not merely sensor noise. As depicted in Figure 6.15, the process is iteratively repeated with a pin attached to the gripper, aiming to reach the center of all the circles. Remarkably, even after 100 iterations, the pin's tip never deviated outside the circle with a diameter of $1mm$.

The findings from the evaluation of precision are as follows:

- The robot's precision and repeatability are as described by Niryo, 0.5 mm.
- At 40% of the maximum velocity, the robot had its most stable z-axis signal.
- For all different velocities, the mean error of the x and y axes, are less than 0.19 mm.



(a) Printed paper with circles of known diameters.

(b) A pointed pin is attached to the gripper and aimed at the center of circles.

FIGURE 6.15: A way used to visualize the precision of the Ned2 cobot.

6.6 Vibration Motor

The Grove vibration motor is utilized in this research to be connected to the gripper, providing vibration during the dosing process as needed. The motor is connected to digital output 4 (DO4) located on the wrist of the cobot, as depicted in Figure 6.16. The connections are shown in the Table 6.2 below.

You can find the full code used to run the vibration motor in Listing 3.5 above in the ROS chapter, as an example of a service server node.

FIGURE 6.16: DO4

Cobot's Wrist	Grove - Vibration Motor
5V	Red - 5V
GND	Black - GND
DO4	Yellow - Signal
DI4	Not connected

TABLE 6.2: The motor connections with the cobot.

Chapter 7

Evaluation & Results

In this chapter, we will evaluate the two experiments conducted in this research and provide a detailed guide for each step of the experimental process.

7.1 Evaluation & Conclusion of Experiment 1

Prior to the execution of Experiment 1, numerous preliminary trials were undertaken to establish a structured approach for the research. The objective of this experiment is to comprehensively grasp the powder's behavior, leading to a conclusive determination of the optimal and efficient dosing methodology for dispensing 1 gram of powder in terms of both time efficiency and quality.

This experiment was conducted 48 times, with each angle and rotational speed tested 4 times. The angles tested were 5°, 13°, 16°, and 21° degrees, and the rotational speeds were 1.775, 1.0, and $0.71 rad/s$. Each trial consisted of a maximum of 100 iterations.

During the preliminary trials, it was observed that the vibration had a significant effect only when the powder was positioned at the edge of the crucible. Additionally, the impact of the vibration was relatively small compared to the influence of the angle and rotation. As a result, **this experiment focused solely on testing the effects of rotation and angle during dosing.**

> **Goal**: Gain a thorough understanding of the characteristics and properties of the powder's mass flow (Sugar in our case).

The steps required to perform Experiment 1 are as follows:

1. Initially, manually fill the glass crucibles with 2 grams $\pm$ 1 mg, as illustrated in Fig. 7.1.
2. After turning the cobot on, building the rosmaster communication as shown in the zero-state in the methodology chapter, calibrating, and loading all the saved trajectories, you start by setting up the balance.
3. Next, we will proceed to pick up the crucible for the dosing process and position it at the specified angle (α), as illustrated in Figure 5.3.

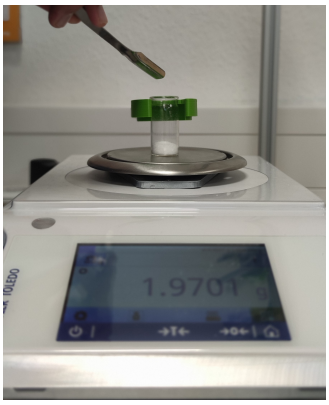


FIGURE 7.1: Manually fill the crucibles.

4. Rotate joint6 from its maximum end (-2.5 radians) to the other maximum end (2.5 radians). By considering the maximum speed, we can determine the duration required (in pulses) for each full rotation by dividing the total duration of 5 radians by the angular velocity in radians per second (ω) to obtain the pulse duration in seconds.
5. After each iteration, the parameters are stored in a dictionary. These parameters include the amount of powder dosed, which is obtained as feedback from the balance, the remaining mass inside the crucible, and the mass flow (which can be calculated using the dosed mass and pulse duration). All this data is saved in a CSV file when the dosed mass reaches or exceeds 1.8g.

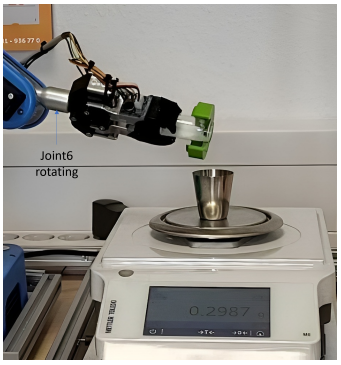
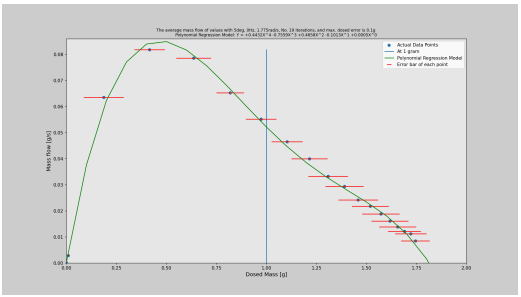
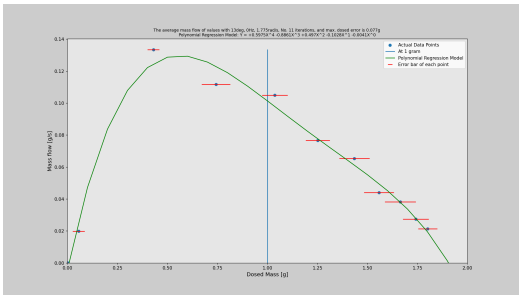


FIGURE 7.2: Illustration of dosing position.

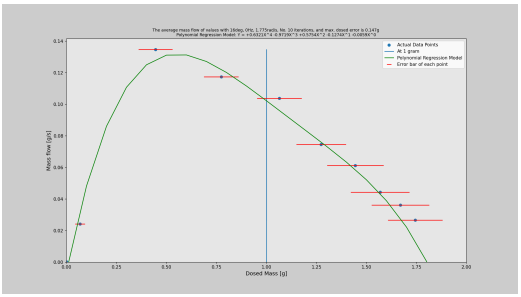
6. The cobot initiates the crucible return process, as described in UML Figure 5.10.
7. Reset the balance.



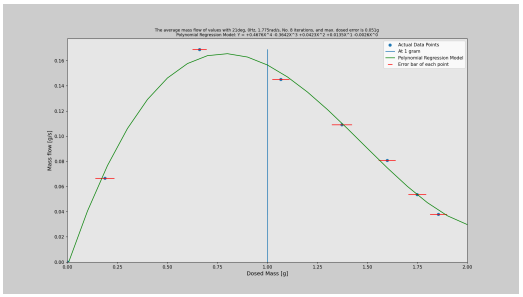
(a) Plot of angle 5° with 19 iterations, Approximated equation: $Y = 0.4432x^4 - 0.756x^3 + 0.466x^2 - 0.1013x + 0.0005$



(b) Plot of angle 13° with 11 iterations, Approximated equation: $Y = 0.5975x^4 - 0.886x^3 + 0.497x^2 - 0.103x - 0.0041$

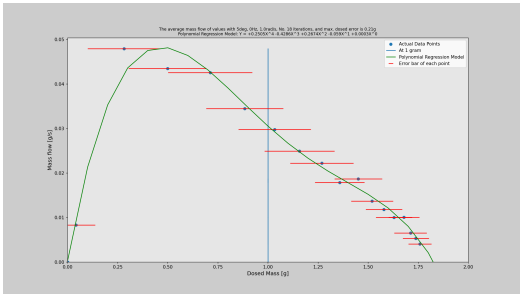


(c) Plot of angle 16° with 10 iterations, Approximated equation: $Y = 0.6321x^4 - 0.972x^3 + 0.5754x^2 - 0.1274x - 0.006$

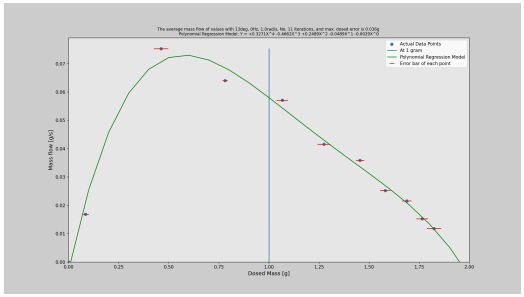


(d) Plot of angle 21° with 8 iterations, Approximated equation: $Y = 0.4676x^4 - 0.3642x^3 + 0.0423x^2 + 0.0135x - 0.0026$

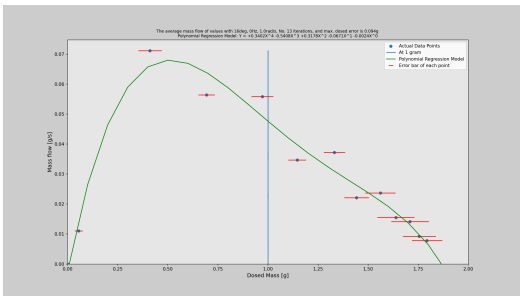
FIGURE 7.3: Comparison between mass flow and dosed mass at full speed rotation ($1.775 rad/s$).



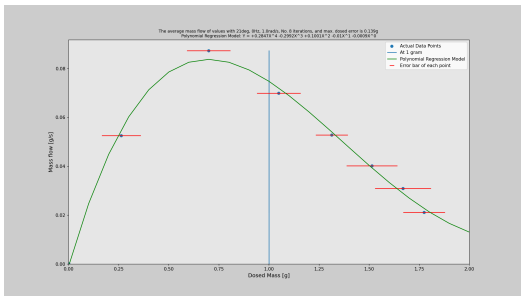
(a) Plot of angle 5° with 18 iterations, Approximated equation: $Y = 0.2505x^4 - 0.4286x^3 + 0.2674x^2 - 0.059x + 0.0003$



(b) Plot of angle 13° with 11 iterations, Approximated equation: $Y = 0.3271x^4 - 0.4662x^3 + 0.249x^2 - 0.049x - 0.0029$



(c) Plot of angle 16° with 13 iterations, Approximated equation: $Y = 0.3402x^4 - 0.5408x^3 + 0.3178x^2 - 0.0671x - 0.0024$



(d) Plot of angle 21° with 8 iterations, Approximated equation: $Y = 0.2847x^4 - 0.2992x^3 + 0.1001x^2 - 0.01x - 0.0009$

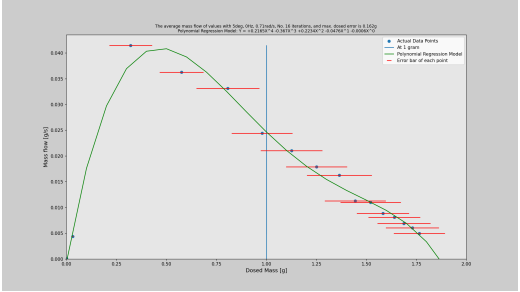
FIGURE 7.4: Comparison between mass flow and dosed mass at around 60% of maximum speed rotation ($1.0 rad/s$).

The results of 48 experiments conducted at various speeds are presented in figures 7.3, 7.4, and 7.5. Each graph represents the average mass flow and dosed mass of four different trials conducted under the same conditions. The red error bar indicates the maximum difference between each iteration across the four trials, while the points represent the average value of each iteration. The blue vertical line at 1 gram serves as a reference for easy analysis of the plot in the next experiment. Lastly, the green polynomial regression function approximates the given data points in each graph.

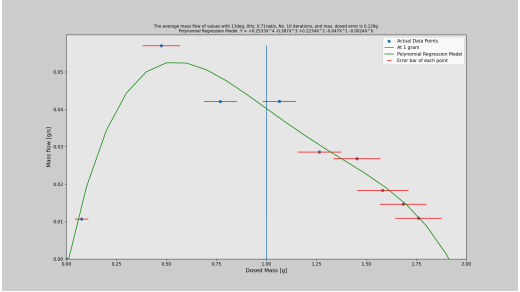
It is worth noting that the behavior of the powder consistently follows a parabolic polynomial regression model, regardless of the angle or speed. This can be attributed to the decreasing mass remaining inside the crucibles as the mass flows through each iteration.

7.1.1 Findings of Experiment 1

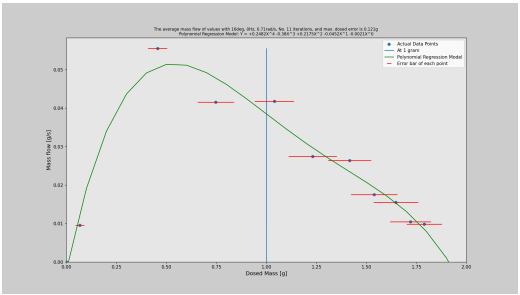
- All the experiments exhibit a consistent parabolic behavior, regardless of the angle and speed of rotation.
- As the angle increases, the number of iterations required to reach 1.8g decreases.
- When the rotational speed is higher, we consistently observe a higher maximum peak of mass flow at the same dosing angle.



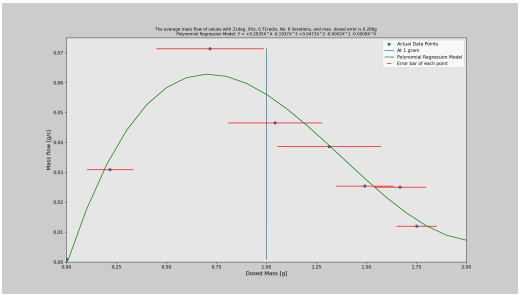
(a) Plot of angle 5° with 16 iterations, Approximated equation: $Y = 0.2165x^4 - 0.367x^3 + 0.2234x^2 - 0.0476x - 0.0006$



(b) Plot of angle 13° with 10 iterations, Approximated equation: $Y = 0.2533x^4 - 0.387x^3 + 0.2234x^2 - 0.047x - 0.0024$



(c) Plot of angle 16° with 11 iterations, Approximated equation: $Y = 0.2482x^4 - 0.38x^3 + 0.2175x^2 - 0.0452x - 0.0021$



(d) Plot of angle 21° with 8 iterations, Approximated equation: $Y = 0.2035x^4 - 0.1937x^3 + 0.0473x^2 - 0.0002x - 0.0009$

FIGURE 7.5: Comparison between mass flow and dosed mass at around 40% of maximum speed rotation ($0.71 rad/s$).

- At 40% of the maximum rotational speed, we observed the highest inconsistency across all angles.
- For angle 5°, it took 4 iterations to be closest to 1 gram of dosed powder and at the same time did not exceed the 1-gram limit for certainty. Similarly, for angles 13° and 16°, it took 3 iterations, and for angle 21°, it took 2 iterations. These iterations ensured that we did not exceed the 1-gram limit.

7.1.2 Conclusion of Experiment 1

- The speed of rotation does not significantly affect the speed of mass flow, as long as there is a noticeable rotation. Rotational speeds above 40% of the maximum velocity are sufficient for the powder dosing task, although the consistency may decrease at lower speeds.
- Rotation and angle manipulation alone may be sufficient for a quick and time-efficient powder dosing task. However, achieving accurate dosage using only these two parameters can be challenging.
- The parabolic behavior occurs as mass flows and the mass inside the crucible diminishes. The initial increase in behavior is a result of the powder not yet reaching the tip of the crucible.

7.2 Evaluation & Conclusion of Experiment 2

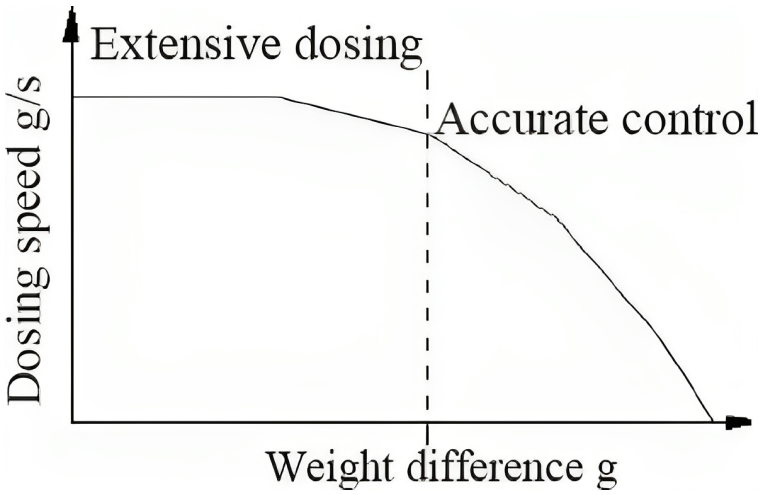


FIGURE 7.6: Division of dosing process [80].

The primary goal of this experiment is to accurately dose 1 gram of sugar in the shortest possible time, ensuring the highest level of precision. Based on the results from Experiment 1, the focus now is on precisely controlling the final portion of dosing to achieve the closest approximation to 1 gram with minimal error.

The experimental approach was inspired by the research conducted by Dejun Zhang, Haochen Wang, and Chunmei Lin in China in 2011 [80]. In their study, they successfully utilized fuzzy control to improve the accuracy of a powder dosing machine. They concluded that the dosing process could be divided into two parts, as shown in Figure 7.6. The first part is the extensive dosing process, while the second part is the accurate control dosing process.

In this experiment, we aim to conclude the extensive dosing process based on our previous experiment. Additionally, we have developed an algorithm to enhance the accuracy of the control dosing process to arrive at 1 gram.

Goal: Investigate the feasibility of accurately measuring and dispensing 1 gram of powdered sugar with minimal error in the shortest possible time.

The steps for performing Experiment 2 are identical to Experiment 1, with the exception of a change in Step 4, and partially in Step 5. Step 4 will be modified as follows:

In case of Angle 5°**:** Perform only four iterations of the rotational procedure used in Experiment 1. As we are certainly below 1 gram but close to it, we change now to only vibration for the accurate control of the dosing process.

```
if dosed <= 0.9:
    duration_of_pulse = 5
    freq = 1
elif (0.9 < dosed <= 0.93):
```

```
    duration_of_pulse = 3
    freq = 1
elif (0.93 < dosed <= 0.96):
    duration_of_pulse = 2
    freq = 1
elif (0.96 < dosed <= 0.99):
    duration_of_pulse = 1
    freq = 1
elif (0.99 < dosed <= 0.9995):
    duration_of_pulse = 1
    freq = 10

```

LISTING 7.1: Algorithm of accurate doing when the angle is 5° or 13°

Angle 13°**:** Rotate the crucible for only 3 iterations. However, if the number of rotations is odd, it is important to ensure that the vibration motor is correctly positioned at the side of the powder, which is located at the bottom of the crucible. This is necessary to ensure that the vibrations are effectively transferred to the powder particles and that they are properly dispensed. The algorithm used for accurate dosage control in the subsequent process is identical to angle 5.

Angle 16°**:** Rotate the crucible for only 3 iterations. Subsequently, the following algorithm is used for the accurate dosage control process:

```
if dosed <= 0.85:
    duration_of_pulse = 3
    freq = 1
elif (0.85 < dosed <= 0.97):
    duration_of_pulse = 1
    freq = 1
elif (0.97 < dosed <= 0.9995):
    duration_of_pulse = 1
    freq = 10

```

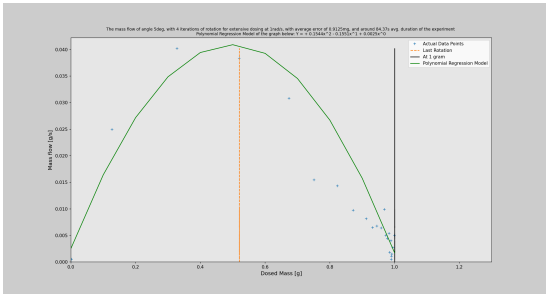
LISTING 7.2: Algorithm of accurate doing when the angle is 16°

Angle 21°**:** Rotate the crucible for only 2 iterations. Subsequently, the following algorithm is used for the accurate dosage control process:

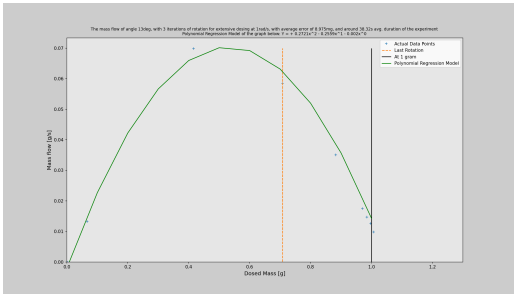
```
if dosed <= 0.87:
    duration_of_pulse = 2
    freq = 1
elif (0.87 < dosed <= 0.97):
    duration_of_pulse = 1
    freq = 1
elif (0.97 < dosed <= 0.9995):
    duration_of_pulse = 1
    freq = 10

```

LISTING 7.3: Algorithm of accurate doing when the angle is 21°

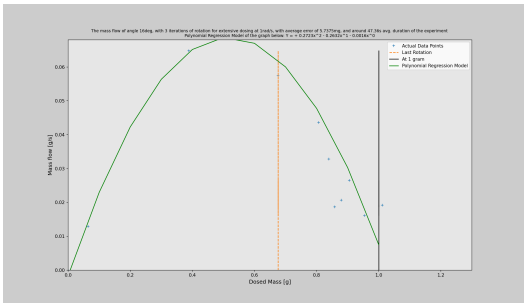
In step 5 of the previous experiment, a modification was implemented in the dosage process. The process is now halted once the mass being dosed reaches or surpasses 0.9995g.



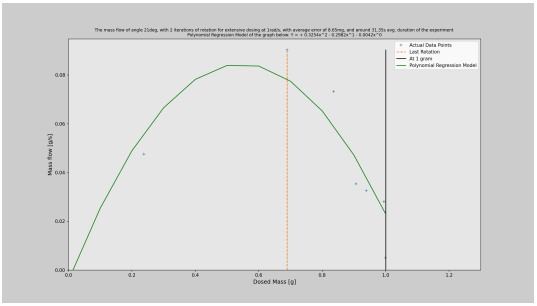
(a) Plot of angle 5° with 4 iterations of rotation, Approximated equation: $Y = 0.1544x^2 - 0.1551x + 0.0025$



(b) Plot of angle 13° with 3 iterations of rotation, Approximated equation: $Y = 0.2721x^2 - 0.2559X - 0.002$



(c) Plot of angle 16° with 3 iterations of rotation, Approximated equation: $Y = 0.2723x^2 - 0.2632X - 0.0016$



(d) Plot of angle 21° with 2 iterations of rotation, Approximated equation: $Y = 0.3254x^2 - 0.2982x - 0.0042$

FIGURE 7.7: Comparison between mass flow and dosed mass at around 60% of maximum speed rotation ($1.0 rad/s$) to reach 1g of dosed mass.

This experiment was conducted in 32 trials, with each angle tested 8 times using the same algorithm. The purpose of this was to test the methodology and ensure its consistency. Figure 7.7 above shows the final trial for each of the four angles. The orange vertical line is drawn at the last rotating iteration, which separates the extensive dosage part of the process (before the line) from the accurate control part (after the line).

7.2.1 Findings of Experiment 2

1. Summary of Angle 5°:
    - Iteration: Varied between 7 and 29, averaging at 18.5 iterations.
    - Duration: Ranged from 44.84 seconds to 117 seconds, with an average of 84.37 seconds (1.41 minutes).
    - Error: Showed a minimum of -0.5 mg, maximum of 4.5 mg, and an average error of 0.91 mg.
2. Summary of Angle 13°:
    - Iteration: Varied between 6 and 12, averaging at 8 iterations.
    - Duration: Ranged from 30.65 seconds to 53.35 seconds, with an average of 38.32 seconds (0.64 minutes).

- Error: Showed a minimum of -0.2 mg, maximum of 29.2 mg, and an average error of 8.98 mg.

3. Summary of Angle 16°:
   - Iteration: Varied between 7 and 17, averaging at 11.5 iterations.
   - Duration: Ranged from 31.51 seconds to 64.73 seconds, with an average of 47.36 seconds (0.79 minutes).
   - Error: Showed a minimum of -0.3 mg, maximum of 13.9 mg, and an average error of 5.74 mg.
4. Summary of Angle 21°:
   - Iteration: Varied between 3 and 16, averaging at 7.5 iterations.
   - Duration: Ranged from 15.55 seconds to 58.81 seconds, with an average of 31.35 seconds (0.52 minutes).
   - Error: Showed a minimum of -0.6 mg, maximum of 20.7 mg, and an average error of 8.65 mg.

7.2.2 Conclusion of Experiment 2

- At an angle of 5°, the minimum errors are observed, with an average of 0.91 mg. However, completing the task at this angle takes the longest time, averaging around a minute and a half.
- At an angle of 13°, the duration time consistently remains very low, averaging 38 seconds across all 8 trails. However, it has the highest recorded errors, with an average of around 9 milligrams.
- At an angle of 16°, it takes a relatively longer amount of time, averaging 47 seconds, to reach a dosage of 1 gram compared to angle 13°. Nevertheless, the average error is significantly lower at around 5.7 mg, which is much less than the error observed at angle 13°.
- At an inclination of 21 degrees, the fastest delivery of the desired dosage is recorded at 15.55 seconds, with an overall average time of 31 seconds. However, due to the high angle of inclination, the accuracy of the control process is compromised, resulting in an average error of 8.65 mg.

7.3 Results Summary

In this chapter, we evaluated two experiments based on different parameters. The first experiment focused on analyzing the behavior of the powder for the extensive dosing process, while the second experiment assessed the error values and duration time for the accurate control of dosing exactly 1 gram. The results indicate that a fully automated powder dosing system can be effectively controlled using a 6-DoF cobot. On average, the system demonstrated a best-case of an average error of 0.9 mg at an angle of 5°, while the worst-case average errors were approximately 9 mg at angles of 13° and 21°. Table 8.1 summarizes the 32 trials conducted in experiment 2.

All the collected data is available in the GitHub repository in Appendix A, or you can refer to some table examples in Appendix B.

Chapter 8

Conclusion & Future Work

In this chapter, we will present a summary and analysis of the methodology and findings of the thesis. We will compare these findings with the objectives outlined in the introduction and the initial expectations based on related work. Furthermore, we will discuss any limitations encountered during the training process. Lastly, in the second section, we will propose recommendations for future research.

8.1 Conclusion

Throughout the course of this thesis, extensive data was collected and documented through over 80 distinct trials and experiments, with 48 for experiment 1 and 32 for experiment 2. Each experiment consisted of a maximum of 100 iterations. Experiment 1 provided a comprehensive understanding of the behavior of the sugar powder, revealing the potential for an extensive powder dosing process through rotational movements. Experiment 2 aimed to test the viability of achieving accurate powder dosage control solely through vibrations. Based on the results, it seems that a 6-DoF cobot has the ability to control a fully automated powder dosing system. The best average precision was 0.9mg, while the least favorable average precision was 9 mg at more inclined angles.

Summary	**Angle** 5°	**Angle** 13°	**Angle** 16°	**Angle** 21°
Duration (s)	Min: 44.84 Max: 117.00 Avg: 84.37	Min: 30.65 Max: 53.35 Avg: 38.32	Min: 31.51 Max: 64.73 Avg: 47.36	Min: 15.55 Max: 58.81 Avg: 31.35
Iterations	Min: 7 Max: 29 Avg: 18.5	Min: 6 Max: 12 Avg: 8.0	Min: 7 Max: 17 Avg: 11.5	Min: 3 Max: 16 Avg: 7.5
Error (mg)	Min: -0.5 Max: 4.5 Avg: 0.91	Min: -0.2 Max: 29.2 Avg: 8.98	Min: -0.3 Max: 13.9 Avg: 5.74	Min: -0.6 Max: 20.7 Avg: 8.65

TABLE 8.1: Experiment summaries for the 4 different angles.

Throughout the 80 trials of both experiments, the motion planning algorithm described in detail in the Methodology chapter has consistently demonstrated a high level of robustness. It has successfully avoided collisions and never encountered any failures. Additionally, let's examine the behavior of each parameter in the independent variables:

Frequency: Vibrations and frequencies have only a minor impact on mass flow compared to angle and rotation, but they are significant when accurately transmitted to the powder at the tip of the crucible.

Angle: Increasing angles lead to greater mass flow.

Rotation: Extensive mass flow is greatly influenced by rotation in general, while the rotational speed does not have a significant impact.

Mass left: The mass left in the crucible is directly proportional to the powder mass flow.

8.2 Future Discussions

In considering future directions for this research, several recommendations and potential improvements emerge:

1. **Diverse Powder Experiments**: A valuable avenue for future research involves replicating the experiments with different powders, such as BOREOX and CELLEOC. This exploration will contribute to a comprehensive understanding of the system's performance across various powder types, potentially uncovering nuances in the dosing process.

2. **Enhanced Controller Algorithm**: To broaden the applicability of the system, enhancements to the controller algorithm are suggested. By incorporating mass flow readings after a single iteration with a 1-second pulse, the algorithm can adapt to diverse dosage processes beyond the 1-gram target, enhancing its versatility.

3. **Machine Learning Integration**: Leveraging the collected data to develop a machine learning model presents an exciting opportunity. Building a refinement model over time ensures that the system evolves and adapts, optimizing its performance based on accumulated insights and experience.

4. **ROS Model Refinement**: Considerable improvement can be achieved by modularizing each process outlined in the methodology section into separate nodes within the ROS model. This approach enhances the system's robustness and dependability, allowing for streamlined troubleshooting and maintenance.

5. **Migration to ROS2**: Exploring the migration to ROS2 is recommended to harness the advantages of DDS (Data Distribution Service) integration. This move ensures secure and encrypted communication between nodes, a critical feature for maintaining the integrity and security of the overall system. The adoption of ROS2 aligns with the evolving landscape of robotic systems and contributes to long-term sustainability.

Appendix A

GitHub Repository - Important Folders & Files

You can access our GitHub repository at the following link: https://github.com/fluxana/FX_Niryo2. Figure A.1 below provides a detailed explanation of all the important folders and files for this project. I chose not to include the full codes and tables in the appendix to avoid making the paper excessively long.

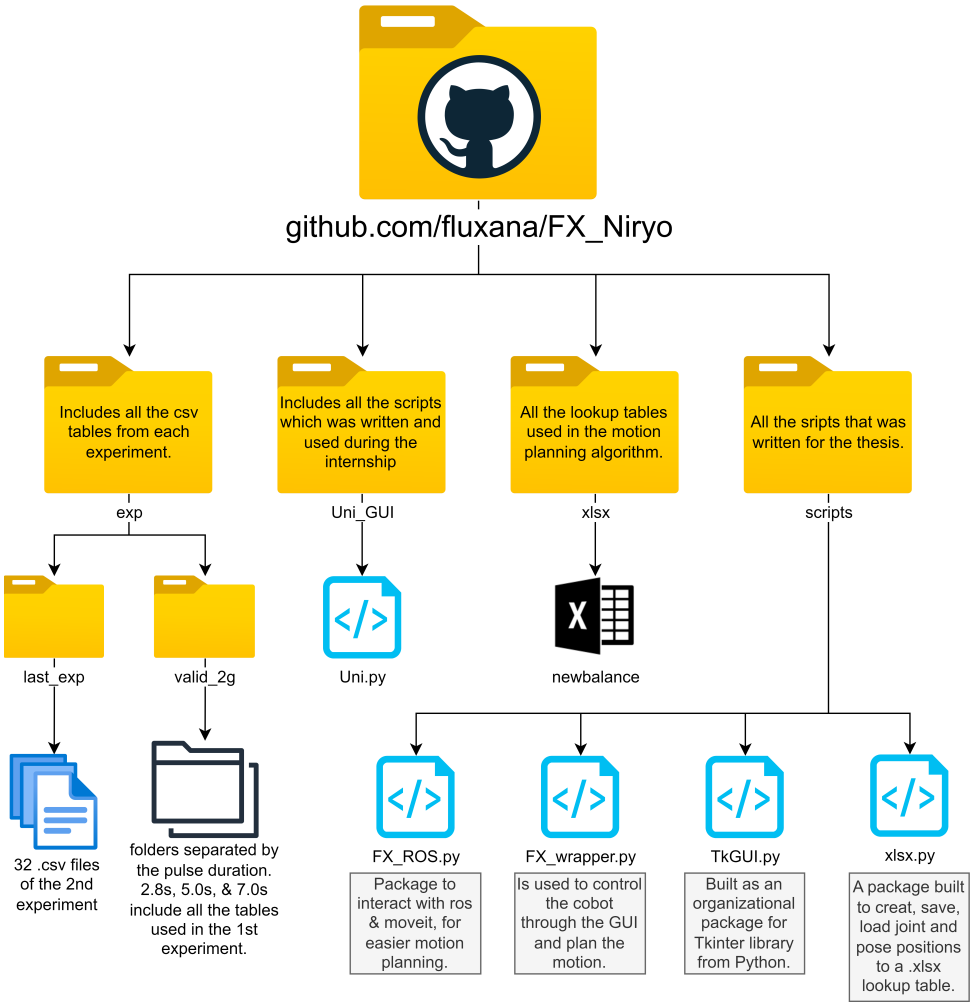


FIGURE A.1: A detailed explanation of the GitHub repository.

Appendix B

Examples of Tabular Data from the last Experiment

The tables typically include the following columns: iterations, timestamp, frequency[Hz], duty_cycle[%], w[rad/s], angle[deg], pulse_duration [s], dosed[g], mass_flow[g/s], and mass_left[g]. The full data can be accessed through the GitHub link provided in Appendix A.

i	timestamp	f[Hz]	w[rad/s]	pulse[s]	dosed[g]	m_left[g]
0	2023-11-22 11:40:57.631384	0	0	0	0.0	2.0
1	2023-11-22 11:41:03.367148	1	1.0	5.0	0.0009	1.9991
2	2023-11-22 11:41:09.063992	1	1.0	5.0	0.1561	1.8439
3	2023-11-22 11:41:14.723656	1	1.0	5.0	0.3712	1.6288
4	2023-11-22 11:41:20.404498	1	1.0	5.0	0.5922	1.4078
5	2023-11-22 11:41:27.765140	1	0	5	0.7841	1.2159
6	2023-11-22 11:41:35.098860	1	0	5	0.8969	1.1031
7	2023-11-22 11:41:42.458512	1	0	5	0.9696	1.0304
8	2023-11-22 11:41:45.560252	1	0	1	0.9824	1.0176
9	2023-11-22 11:41:48.649260	1	0	1	0.995	1.005
10	2023-11-22 11:41:51.697005	2	0	1	0.9965	1.0035
11	2023-11-22 11:41:54.740041	10	0	1	0.9965	1.0035
12	2023-11-22 11:41:57.773799	10	0	1	0.9975	1.0025
13	2023-11-22 11:42:00.835649	10	0	1	0.9975	1.0025
14	2023-11-22 11:42:03.873502	10	0	1	0.9975	1.0025
15	2023-11-22 11:42:06.942339	10	0	1	0.9976	1.0024
16	2023-11-22 11:42:10.004315	10	0	1	0.9981	1.0019
17	2023-11-22 11:42:13.074265	10	0	1	0.9981	1.0019
18	2023-11-22 11:42:16.154055	10	0	1	0.9982	1.0018
19	2023-11-22 11:42:19.200897	10	0	1	1.0003	0.9997

TABLE B.1: 5^{th} Trial of the 2^{nd} experiment for angle 5°.

i	timestamp	f[Hz]	w[rad/s]	pulse[s]	dosed[g]	m_left[g]
0	2023-11-22 14:46:42.743219	0	0	0	0.0001	2.0
1	2023-11-22 14:46:48.425035	1	1.0	5.0	0.0664	1.9336
2	2023-11-22 14:46:54.153774	1	1.0	5.0	0.4156	1.5844
3	2023-11-22 14:46:59.860531	1	1.0	5.0	0.7078	1.2922
4	2023-11-22 14:47:07.239273	1	0	5	0.8829	1.1171
5	2023-11-22 14:47:14.559944	1	0	5	0.9703	1.0297
6	2023-11-22 14:47:17.652783	1	0	1	0.985	1.015
7	2023-11-22 14:47:20.739554	1	0	1	0.9976	1.0024
8	2023-11-22 14:47:23.832531	1	0	1	1.0074	0.9926

TABLE B.2: Last Trial of the 2^{nd} experiment for angle 13°.

i	timestamp	f[Hz]	w[rad/s]	pulse[s]	dosed[g]	m_left[g]
0	2023-11-22 12:38:55.860314	0	0	0	0.0	2.0
1	2023-11-22 12:39:01.547093	10	1.0	5.0	0.0884	1.9116
2	2023-11-22 12:39:07.267854	10	1.0	5.0	0.4285	1.5715
3	2023-11-22 12:39:12.990604	10	1.0	5.0	0.7755	1.2245
4	2023-11-22 12:39:18.196383	1	0	3	0.8585	1.1415
5	2023-11-22 12:39:23.411143	1	0	3	0.9254	1.0746
6	2023-11-22 12:39:26.507193	1	0	1	0.9457	1.0543
7	2023-11-22 12:39:29.597848	1	0	1	0.9573	1.0427
8	2023-11-22 12:39:32.683757	1	0	1	0.9638	1.0362
9	2023-11-22 12:39:35.775669	1	0	1	0.9711	1.029
10	2023-11-22 12:39:38.869494	1	0	1	0.9805	1.0195
11	2023-11-22 12:39:41.956353	1	0	1	0.9891	1.0109
12	2023-11-22 12:39:45.047286	1	0	1	0.9997	1.0003

TABLE B.3: 5^{th} Trial of the 2^{nd} experiment for angle 16°.

i	timestamp	f[Hz]	w[rad/s]	pulse[s]	dosed[g]	m_left[g]
0	2023-11-22 15:04:33.401463	0	0	0	0.0	2.0
1	2023-11-22 15:04:39.097239	1	1.0	5.0	0.2376	1.7624
2	2023-11-22 15:04:44.820979	1	1.0	5.0	0.689	1.311
3	2023-11-22 15:04:48.976900	1	0	2	0.8356	1.1644
4	2023-11-22 15:04:53.485636	1	0	2	0.9062	1.0938
5	2023-11-22 15:04:56.578433	1	0	1	0.9387	1.0613
6	2023-11-22 15:05:00.717235	1	0	2	0.9948	1.0052
7	2023-11-22 15:05:03.788211	2	0	1	0.9998	1.0002

TABLE B.4: Last Trial of the 2^{nd} experiment for angle 21°.

Appendix C

Important Python Fuctions Used in this Project

C.1 Calling a Service Server

As shown in the code below, rather than

```
def Call_Aservice(service_name, type, request_name=None,req_args=None):
    """Call a ROS service.

    Parameters:
    ......................
    service_name: str
    type: srv
    request_name: None (srv)
    req_args: None (dictionary) ex. {'positon': 210, 'id': 11, 'value':
     False}

    Returns:
    ......................
    Return the response of the service.
    """
    try:
        rospy.wait_for_service(service_name, 2)
    except (rospy.ServiceException, rospy.ROSException) as e:
        rospy.logerr("Timeout and the Service was not available : " +
    str(e))
        return RobotState()

    try:
        service_call = rospy.ServiceProxy(service_name, type)
        if request_name == None:
            response = service_call()
        else:
            request = request_name()
            for key, value in req_args.items():
                #print("f{key} = {value}")
                method = setattr(request, key, value)
            response = service_call(request)
    except rospy.ServiceException as e:
        rospy.logerr("Falied to call the Service: " + str(e))
        return 0

    return response
```

LISTING C.1: Call_Aservice Function

C.1.1 Example of Call_Aservice Function

The code snippet presented below serves as an illustration of the ease with which the Call_Aservice function is employed. This example specifically functions as a service client aimed at calibrating the motors of the Ned2 robot, a crucial step preceding any motion planning. Refer to section 5.3.2 for comprehensive insights into the motor calibration process.

```
def motor_cal():
    cal_service = '/niryo_robot/joints_interface/calibrate_motors'
    Call_Aservice(cal_service, type=SetInt, request_name=SetIntRequest,
     req_args={"value":1})
```

LISTING C.2: motor_cal Function

C.2 Subscribing to Topic

```
def Subscribe(topic_name, type, msg_args):
    """Subscribe to a certain topic.

    Parameters:
    ......................
    topic_name: str
    type: srv
    msg_args: list >> list of strings, which contains the arguments
    that we need to read from the topic.

    Returns:
    ......................
    Return a list of the read values from each argument.
    If we have only one argument, it returns the value of this argument
     only, not a list.
    """
    try:
        msg = rospy.wait_for_message(topic_name, type, 2)
    except:
        rospy.logerr("Timeout and the Topic Did not recieve any
    messages")
        return 0

    value = []
    if len(msg_args) == 1:
        value = getattr(msg, msg_args[0])
    else:
        for i in msg_args:
            value.append(getattr(msg, i))

    return value
```

LISTING C.3: Subscribe Fucntion

C.2.1 Examples of Subscribe Function

The subsequent functions demonstrate the straightforward process of subscribing to a particular topic and retrieving the intended data. In the first case, the topic "/joint_states" acquires its published information concerning the present joint positions from the sensors node. Furthermore, the get_pose

function is designed to subscribe to the "/nirio_robot/robot_state" topic, which disseminates a RobotState message.

```
def Get_joints():
    """return a tuple of 6 values for each joint from 1 till 6"""

    joints_values = Subscribe('/joint_states', JointState, ["position"
    ])
    return joints_values
```

LISTING C.4: Get_joints Function

```
def get_pose():
    """Gets the pose values from the robot_state topic.
    Return:
    ..................................
    a list of two dictionaries, the first is positions (x,y,z),
    whereas the second is the rpy (roll, pitch, yaw)
    """

    return Subscribe('/niryo_robot/robot_state', RobotState, ['position
    ', 'rpy'])
```

LISTING C.5: get_pose Function

C.3 Moveing the Cobot using Joints

```
arm = moveit_commander.move_group.MoveGroupCommander("arm")
```

```
def move_to_joints(joints, arm_speed=None):
    """Move to a given joint values.
    Parameters:
    .........................

    joints: list or tuble -> [joint1, joint2, joint3, joint4, joint5,
    joint6]
    arm_speed: float (optional) -> between 0 and 1. (0,1]
    """
    joints_limits = Get_Joints_limits()

    for i in range(6):
        if joints_limits.joint_limits[i].max < joints[i] or joints[i] <
     joints_limits.joint_limits[i].min:
            print("Joint{} = {}, which is out of limit!".format(i+1,
    joints[i]))
            print("Joint{} can not be more than {} neither less than {}
    ".format(i+1, joints_limits.joint_limits[i].max, joints_limits.
    joint_limits[i].min))
            return
        else:
            pass

    #arm.set_joint_value_target(joints)
    if arm_speed:
        set_speed(arm_speed)
    arm.go(joints, wait=True)

    arm.stop()
```

LISTING C.6: Moving to a certain joints values

```python
def Move_joint_axis(axis, new=None, add=None, arm_speed=None):
    """You should either put a value to add or new, not both.

    Parameters:
    ......................
    * axis: int -> the number of the joint that you want to move

    * new: float -> The new coordination you want to give to a joint (axis).
           "new" will always overright the value of the axis.
    * add: float -> the value in meters change in a certain joint (axis).
    * arm_speed: float (optional) -> between 0 and 1. (0,1]

    Returns: None
    ......................
    """
    moving_joints = list(Get_joints())

    if new:
        moving_joints[axis-1] = new
    elif add:
        moving_joints[axis-1] += add

    joints_limits = Get_Joints_limits()

    if joints_limits.joint_limits[axis-1].max < moving_joints[axis-1] or moving_joints[axis-1] < joints_limits.joint_limits[axis-1].min:
        print("The joint{} can not be more than {} neither less than {}".format(axis, joints_limits.joint_limits[axis-1].max, joints_limits.joint_limits[axis-1].min))
        return 0
    else:
        pass

    arm.set_joint_value_target(moving_joints)
    if arm_speed:
        set_speed(arm_speed)
    arm.go(moving_joints, wait=True)

    arm.stop()
```

LISTING C.7: Moving only one axis in joints

C.4 Moveing the Cobot using Pose

```python
def Move_to_pose(pose_values, arm_speed=None):
    """Move to a given pose values.
    Parameters:
    ......................

    pose_values: list or tuble -> [x, y, z, roll, pitch, yaw]
    arm_speed: float (optional) -> between 0 and 1. (0,1]
    """

    pose = Pose()
    p_goal = pose.position
    orn_goal = pose.orientation

    p_goal.x = pose_values[0]
```

```
    p_goal.y = pose_values[1]
    p_goal.z = pose_values[2]

    roll = pose_values[3]
    pitch = pose_values[4]
    yaw  = pose_values[5]

    orn_goal.x, orn_goal.y, orn_goal.z, orn_goal.w = tf.transformations
    .quaternion_from_euler(roll,pitch,yaw)

    #arm.set_goal_tolerance(0.001)
    if arm_speed:
        set_speed(arm_speed)
    arm.set_pose_target(pose)
    arm.go(wait=True)

    arm.stop()
    arm.clear_pose_targets()
```

LISTING C.8: Moving to certain pose values of the TCP

```
def Move_pose_axis(axis, new=None, add=None, arm_speed=None):
    """You should either put a value to add or new, not both.

    Parameters:
    ......................
    * axis: str -> (x, y, z, roll, pitch, or yaw)
    * new: float -> The new coordination you want to give to a certain
    axis.
           "new" will always overwrite the value of the axis.
    * add: float -> the value in meters or radians you want to add to a
     certain axis.
    * arm_speed: float (optional) -> between 0 and 1. (0,1]
    Returns: None
    ......................
    """
    FK = get_pose()
    axises = ['x','y','z']

    pose = Pose()
    p_goal = pose.position
    orn_goal = pose.orientation

    p_current = FK[0]

    rpy_current = FK[1]

    if add:
        if axis.lower() in axises:
            current_value = getattr(p_current, axis)
            setattr(p_current, axis, current_value+add)
        else:
            current_value = getattr(rpy_current, axis)
            setattr(rpy_current, axis, current_value+add)
    if new:
        if axis.lower() in axises:
            setattr(p_current, axis, new)
        else:
            setattr(rpy_current, axis, new)


    p_goal.x = p_current.x
    p_goal.y = p_current.y
```

```
41      p_goal.z = p_current.z
42
43      orn_goal.x, orn_goal.y, orn_goal.z, orn_goal.w = tf.transformations
        .quaternion_from_euler(rpy_current.roll,rpy_current.pitch,
        rpy_current.yaw)
44
45      arm.set_pose_target(pose)
46
47      if arm_speed:
48          set_speed(arm_speed)
49      arm.go(wait=True)
50
51      arm.stop()
52      arm.clear_pose_targets()
```

LISTING C.9: Moving only one axis in Pose

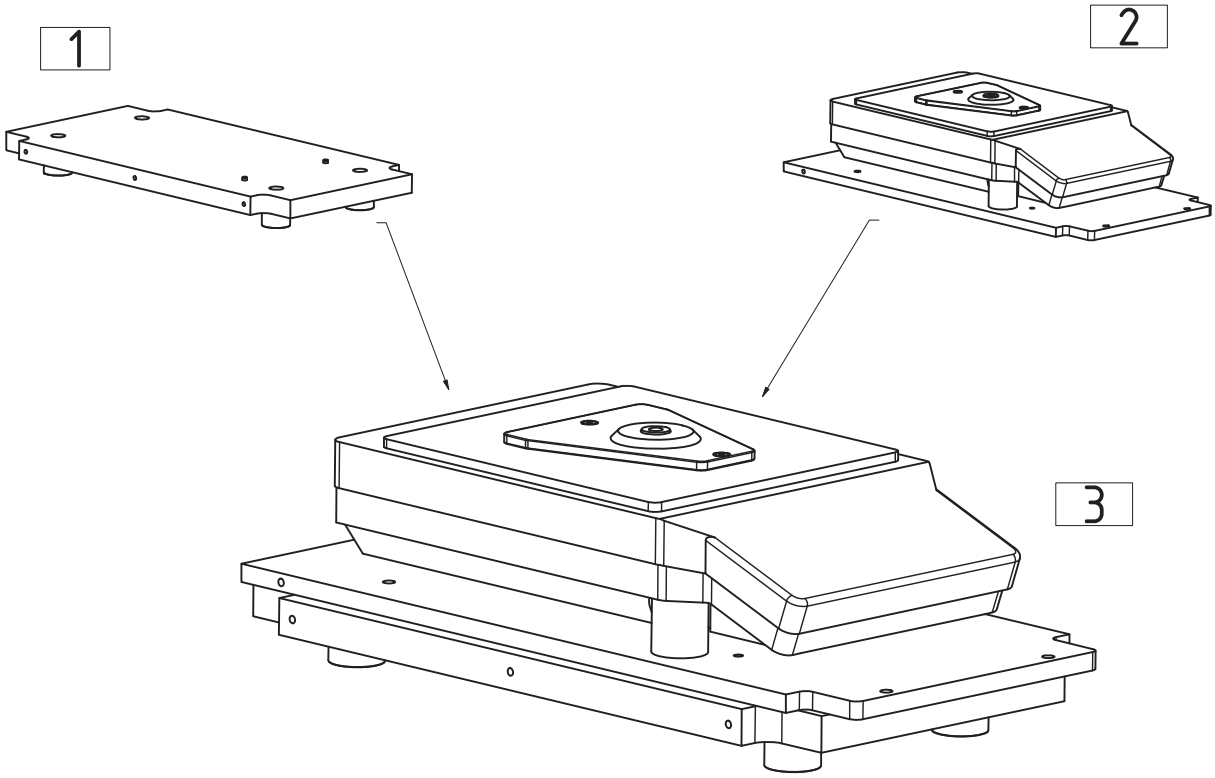
C.5 Use Forward Kinematics

```
1  def FK_Moveit(joints):
2      """Get Forward Kinematics from the MoveIt service directly after
       giving joints
3      :param  joints
4      :type   joints: list of joints values
5      :return: A Pose state object
6      @example of a return
7
8  position:
9    x: 0.278076372862
10   y: 0.101870353599
11   z: 0.425462888681
12 orientation:
13   x: 0.0257527874589
14   y: 0.0122083384395
15   z: 0.175399274203
16   w: 0.984084775322
17
18     """
19     rospy.wait_for_service('compute_fk', 2)
20     moveit_fk = rospy.ServiceProxy('compute_fk', GetPositionFK)
21
22     fk_link = ['base_link', 'tool_link']
23     header = Header(0, rospy.Time.now(), "world")
24     rs = RobotStateMoveIt()
25
26     rs.joint_state.name = ['joint_1', 'joint_2', 'joint_3', 'joint_4',
       'joint_5', 'joint_6']
27     rs.joint_state.position = joints
28
29     reponse = moveit_fk(header, fk_link, rs)
30
31     return reponse.pose_stamped[1].pose
```

LISTING C.10: Using forward kinematics to get the pose coordination from MoveIt

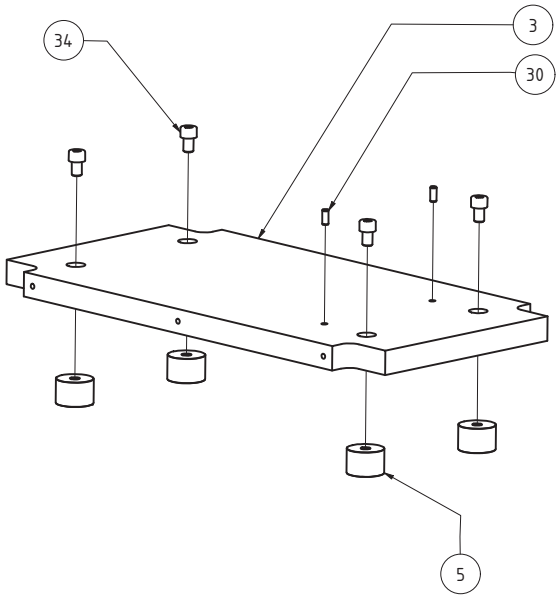
Appendix D

Balance Technical Drawing

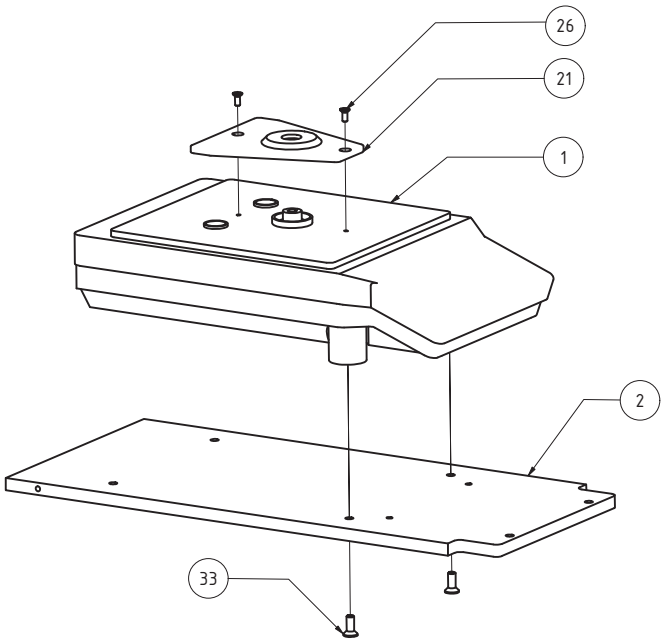


Fluxana GmbH & Co.KG			Datum	Name
		Gezeichn.	20.02.2019	J.Schramm
Bezeichnung: Schublade mit Waage		Kontroll.		
		Norm		
Projekt: Boramat 18 & 30				1 /3
Baugruppen-Nr. BG0106	Version: 2	Art.-Nr.		A3

Schritt 1: Grundplatte Schublade und Gummifüße

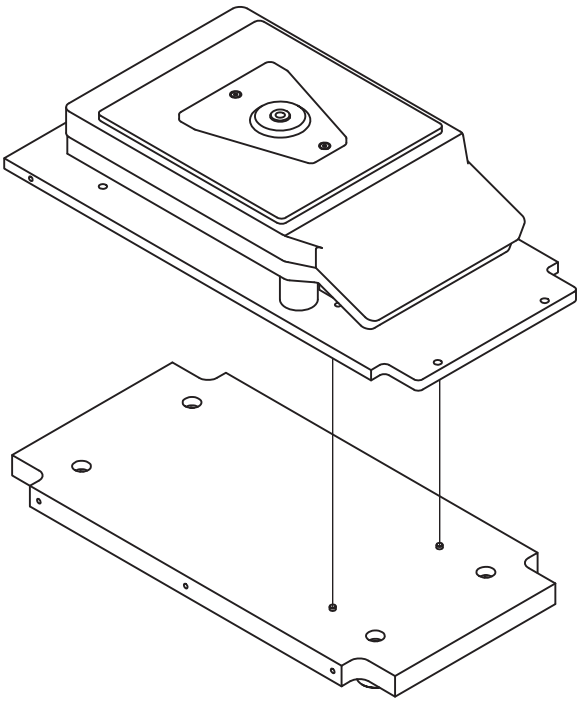


Schritt 2: Grundplatte Schublade und Analysenwaage



Fluxana GmbH & Co.KG			Datum	Name
		Gezeichn.	20.02.2019	J.Schramm
Bezeichnung: Schublade mit Waage		Kontroll.		
		Norm		
Projekt: Boramat 18 & 30				2 /3
Baugruppen-Nr. BG0106	Version: 2	Art.-Nr.		A3

Schritt 3: Montage der Führungsschienen

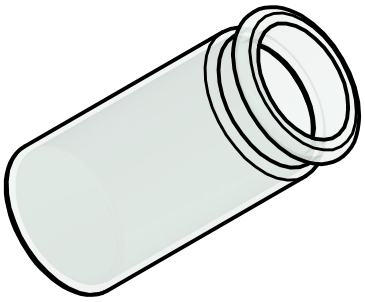


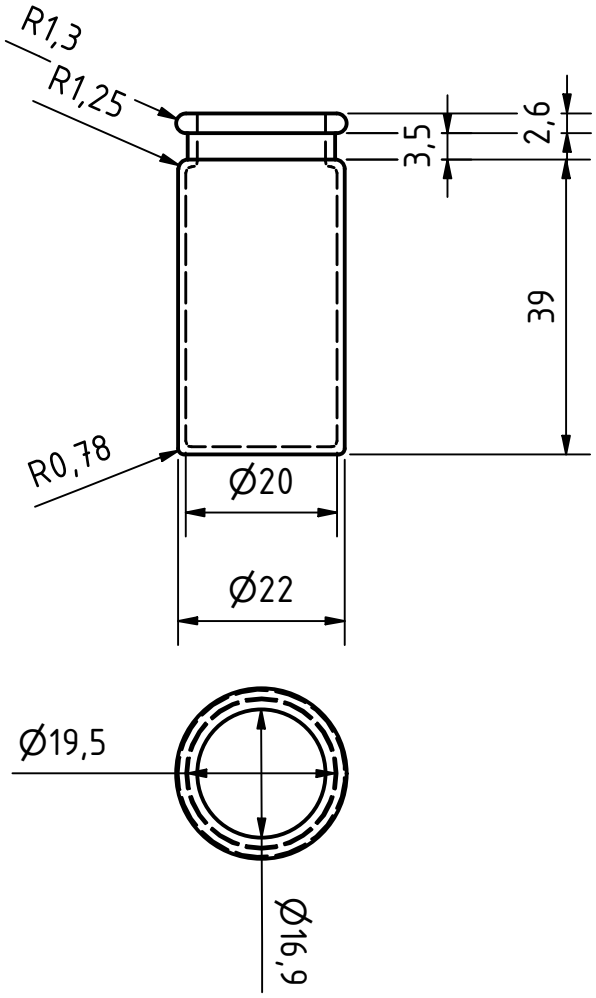
PARTS LIST			
ITEM	QTY	PART NUMBER	DESCRIPTION
1	1		Mettler Toledo Waage
2	1	BT0413	Grundplatte Waage
3	1	BT0414	Grundplatte Schublade
5	4		Gummipuffer 30x20mm M8x8
21	1	BT0419	Schutz Waage
26	2	DIN EN ISO 10642	Senkschraube mit Inbus M4 x 10
30	2	ISO 2338	Zylinderstifte 5 m6 x 12
33	2	DIN EN ISO 10642	Senkschraube mit Inbus M6 x 16
34	4	DIN EN ISO 4762	Zylinderschraube mit Inbus M8 x 12

Fluxana GmbH & Co.KG		Datum	Name
	Gezeichn.	20.02.2019	J.Schramm
Bezeichnung: Schublade mit Waage	Kontroll.		
	Norm		
Projekt: Boramat 18 & 30			3 /3
Baugruppen-Nr. BG0106	Version: 2	Art.-Nr.	A3

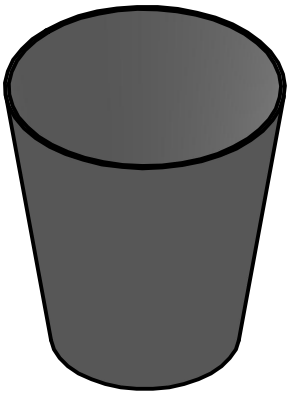
Appendix E

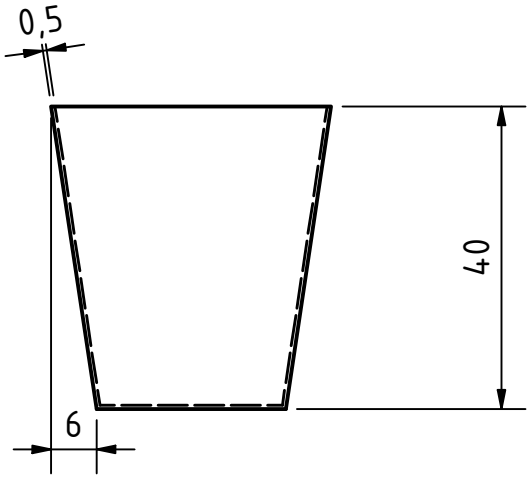
Glass and Metal Crucibles Technical Drawing

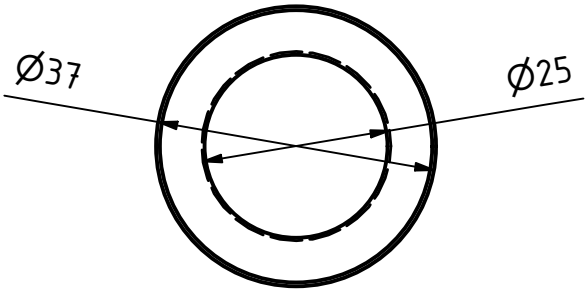




Allgemeintoleranz: DIN ISO 2768-mK			Alle Kanten R0,1 - 0,3 entgratet	
Erstellt durch: Abdelrahman Mostafa	Genehmigt von:	Gewicht: N/A		Werkstoff: Glas
Bauteilbezeichnung:				
FLUXANA® XRF Application Solutions	Projekt:		Artikelnr.:	
			Bauteilzeichnungsnr.: Schnappdeckelglas-1	
			Format: A4 / Maßstab: 1 : 1 / Änd.: 0 / Datum: 24/10/2023 / Spr.: de / Blatt: 1/1	







Allgemeintoleranz: DIN ISO 2768-mK			Alle Kanten R0,1 - 0,3 entgratet	
Erstellt durch: Abdelrahman Mostafa	Genehmigt von:	Gewicht: N/A		Werkstoff: Generisch

Bauteilbezeichnung:

FLUXANA®
XRF Application Solutions

Projekt:

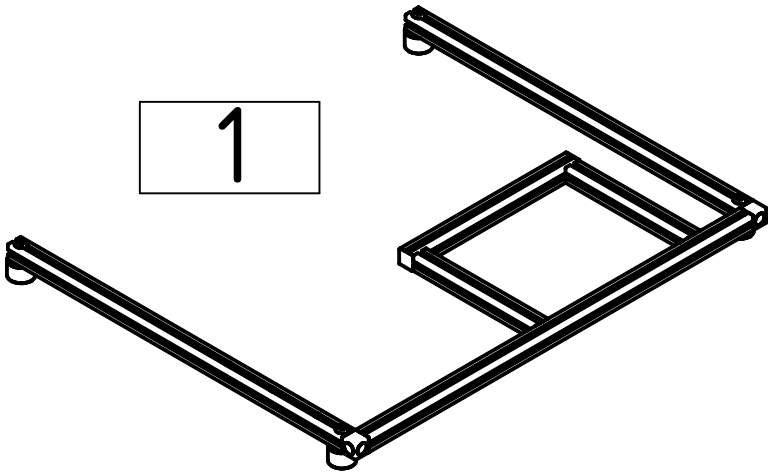
Artikelnr.:

Bauteilzeichnungsnr.: Boromat_tiegel-1

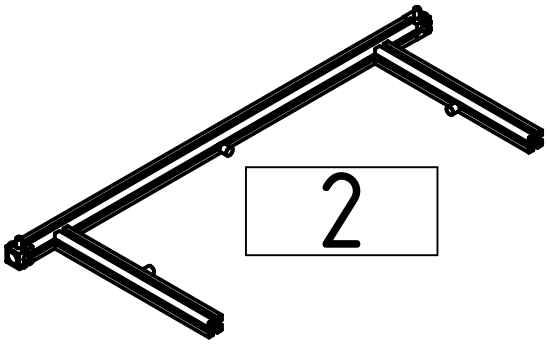
Format	Maßstab:	Änd.:	Datum:	Spr.:	Blatt
A4	1 : 1	0	24/10/2023	de	1/1

Appendix F

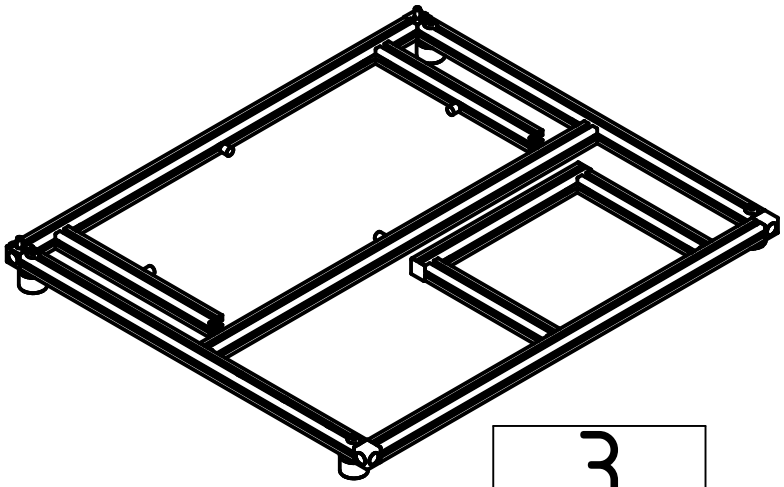
Frame Technical Drawing



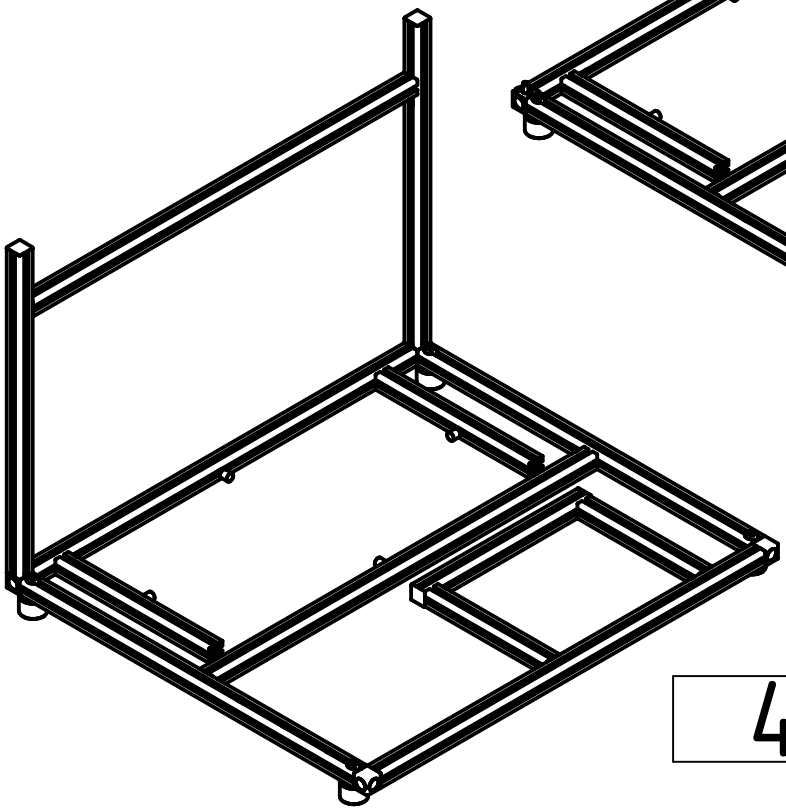
1



2



3



4

Erstellt durch:	Genehmigt von:	Gewicht:	
Carine Allen		N/A	

Montagebaugruppenbez.: Niryo Frame

FLUXANA®
XRF Application Solutions

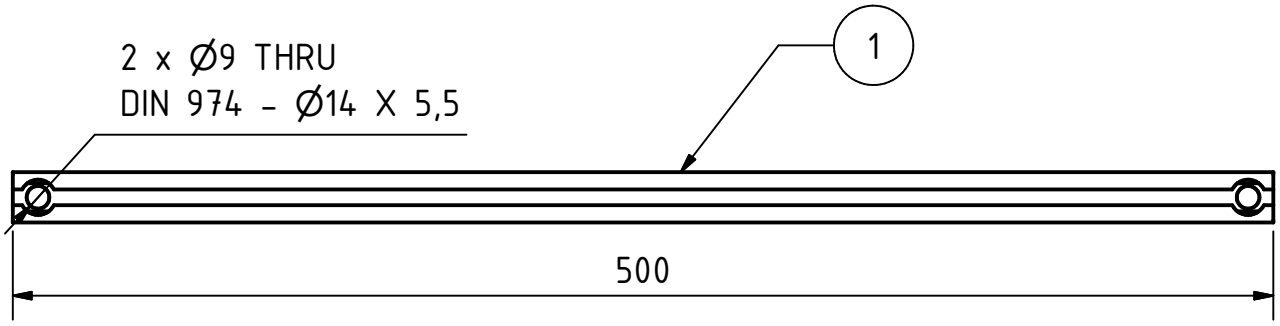
Projekt:

Artikelnr.:

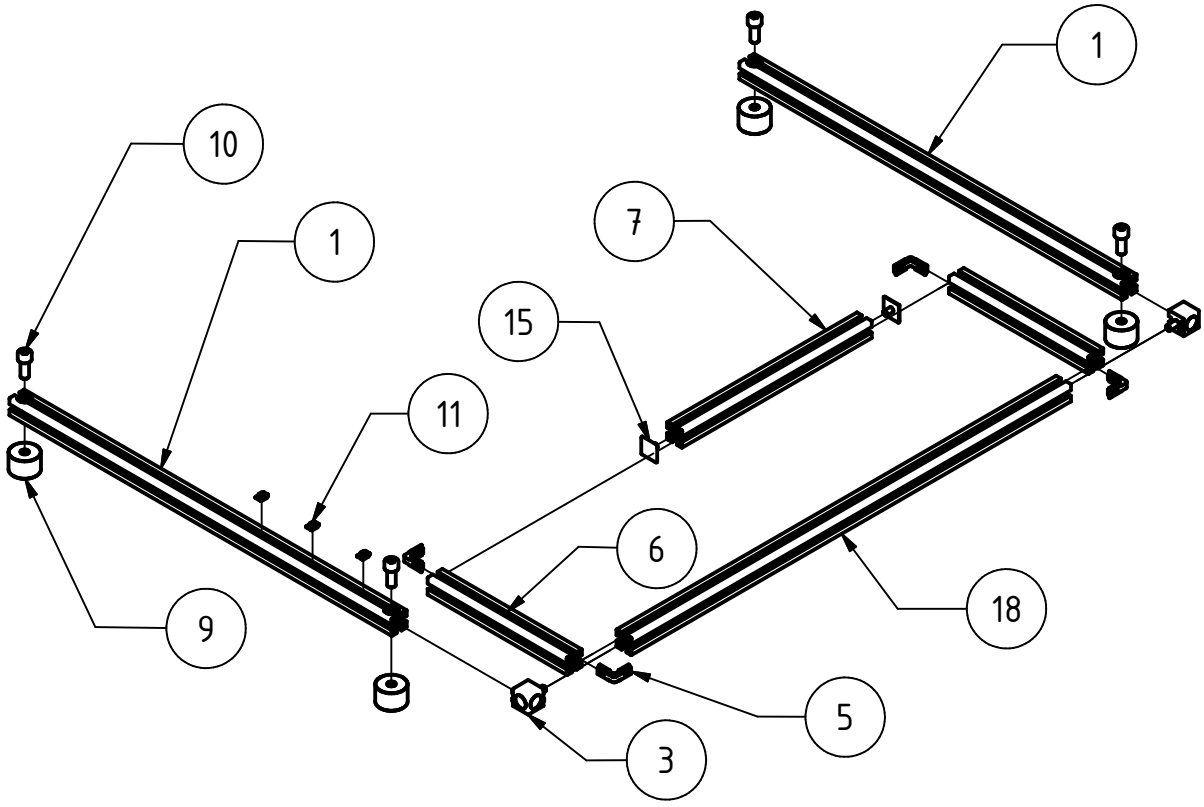
Montagebaugruppenzeichnungsnr.:

Format:	Maßstab:	Änd.:	Datum:	Spr.:	Blatt
A4	1 : 8	0	21/06/2023	de	1/5

Punkt 1: 2 x Bohrung im Profil 20x200x500mm



Schritt 1: Niryo holder



Erstellt durch:	Genehmigt von:	Gewicht:	
Carine Allen		N/A	

Montagebaugruppenbez.: Niryo Frame

FLUXANA®
XRF Application Solutions

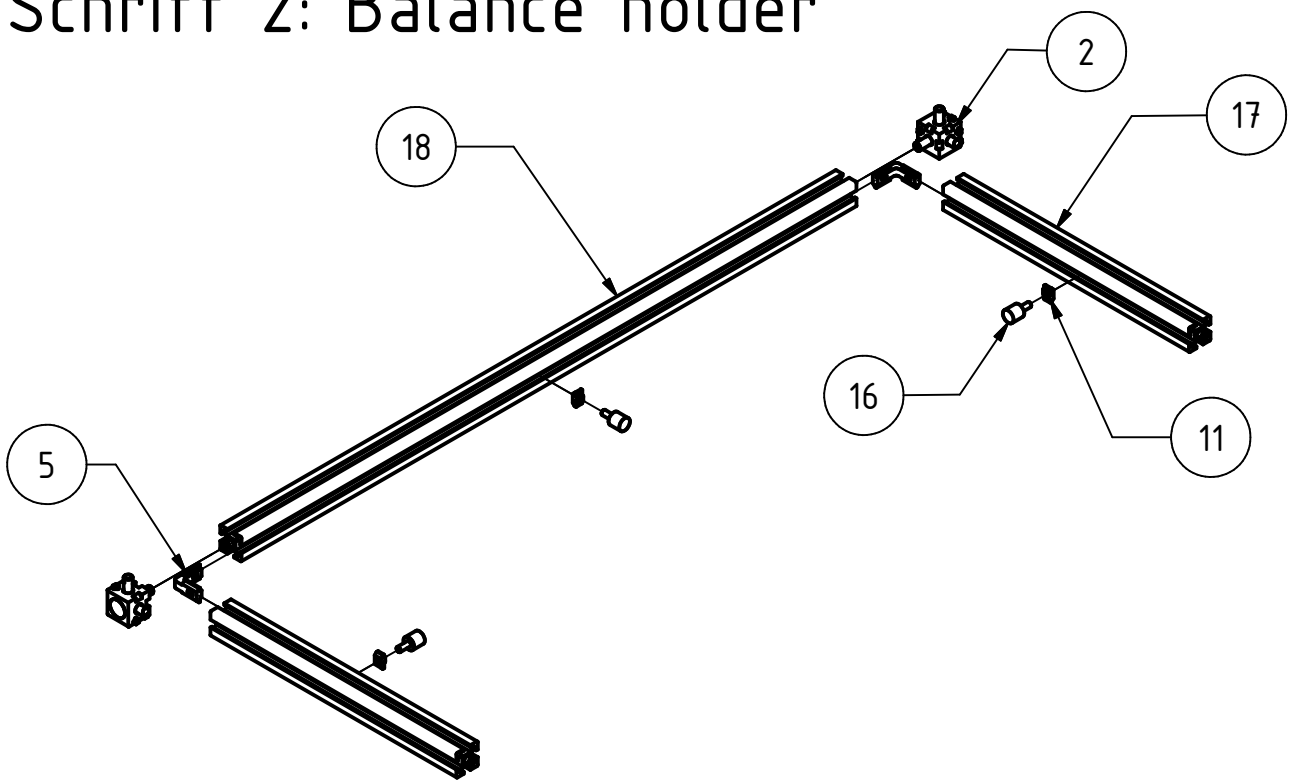
Projekt:

Artikelnr.:

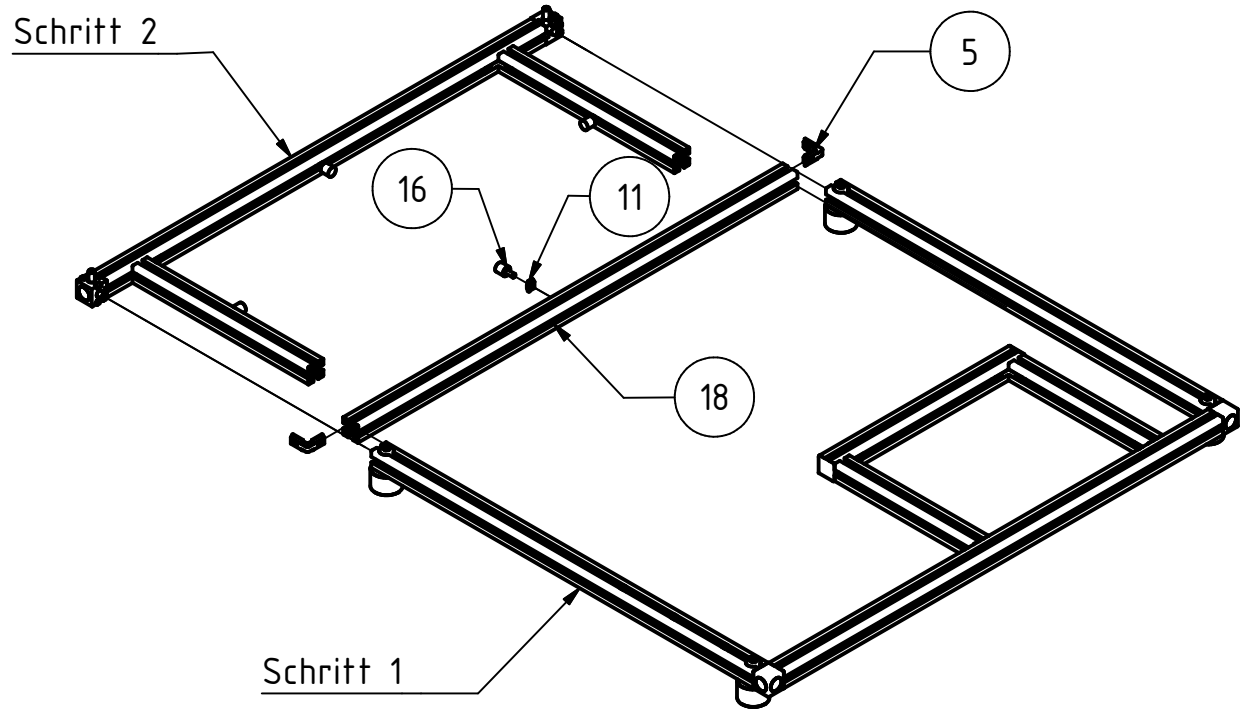
Montagebaugruppenzeichnungsnr.:

Format:	Maßstab:	Änd.:	Datum:	Spr.:	Blatt
A4	1 : 7	0	21/06/2023	de	2/5

Schritt 2: Balance holder



Schritt 3: Base frame



Erstellt durch:	Genehmigt von:	Gewicht:	
Carine Allen		N/A	

Montagebaugruppenbez.: Niryo Frame

FLUXANA®
XRF Application Solutions

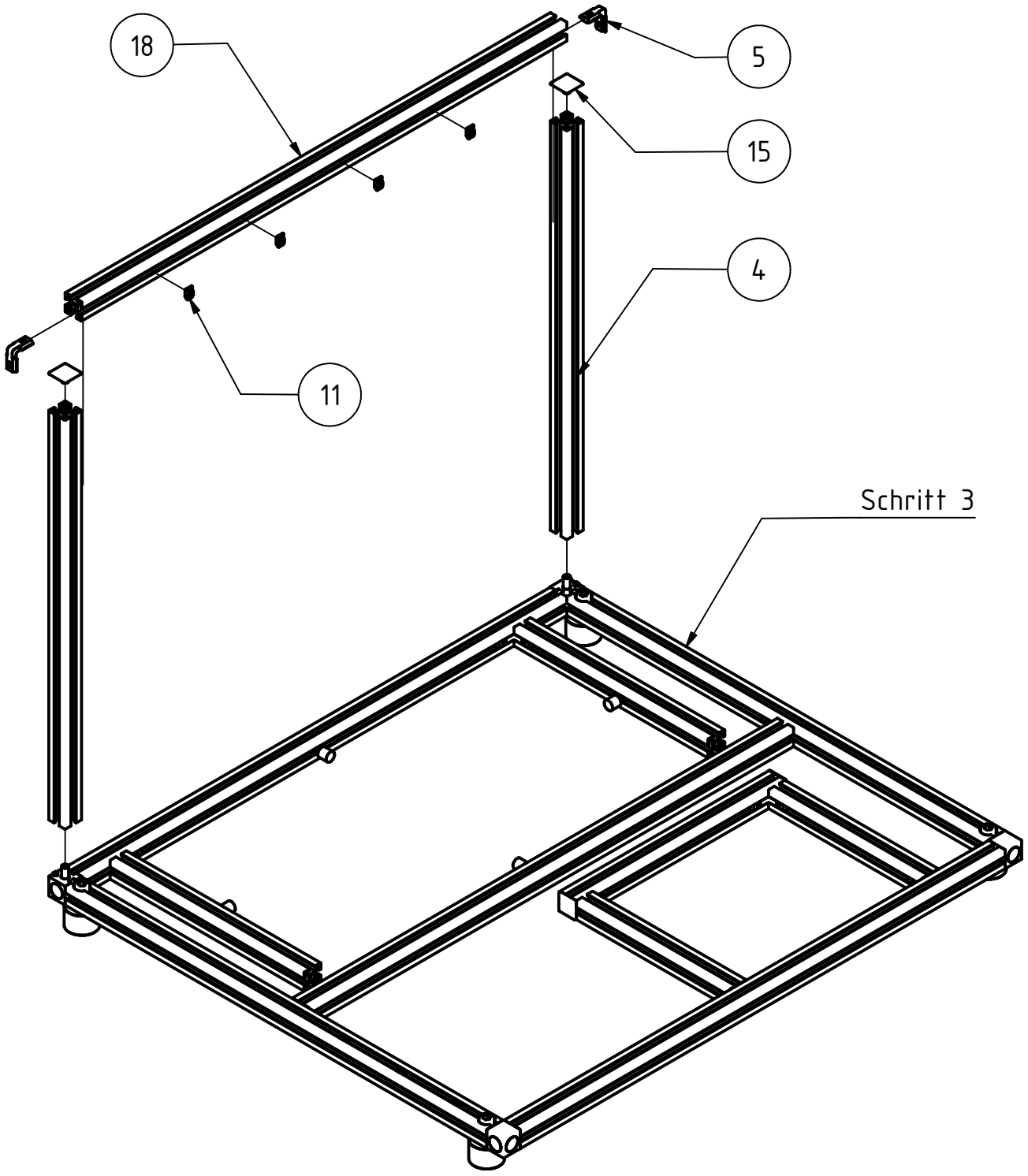
Projekt:

Artikelnr.:

Montagebaugruppenzeichnungsnr.:

Format:	Maßstab:	Änd.:	Datum:	Spr.:	Blatt
A4	1 : 5	0	21/06/2023	de	3/5

Schritt 4: Complete frame



Erstellt durch:	Genehmigt von:	Gewicht:	
Carine Allen		N/A	

Montagebaugruppenbez.: Niryo Frame

FLUXANA®
XRF Application Solutions

Projekt:

Artikelnr.:

Montagebaugruppenzeichnungsnr.:

Format:	Maßstab:	Änd.:	Datum:	Spr.:	Blatt
A4	1 : 5	0	21/06/2023	de	4/5

Montagebaugruppenstückliste

Pos.	Anz.	Artikelnr.	Norm-/ BT-nr	Beschreibung
1	2	VI-0342		Strebenprofil 20x20x500mm
2	2	VI-0348		Würfelverbinder 20/3
3	2	VI-0347		Würfelverbinder 20/2
4	2	VI-0342		Strebenprofil 20x20x420mm
5	10	VI-0344		Innenwinkel 6R
6	2	VI-0342		Strebenprofil 20x20x182mm
7	1	VI-0342		Strebenprofil 20x20x248mm
8	1		BG0106	Schublade mit Waage
9	4	BO-0222		Gummipuffer 30x20mm M8x8
10	4		ISO 4762 - M8 x 20	Hexagon Socket Head Cap Screw
11	11	KT-06-0012		Hammermutter 6 M4
12	4		Becherglass_hanger	
13	1		Schnappdeckelglas	
14	1	BO-0289	BT0953	Adapterring Schnappdeckelglas Boramat 30
15	4	KT-07-0001		Abdeckkappe grau 20x20
16	4	GR-0186		Gummipuffer D10 x H10 M4x10AES
17	2	VI-0342		Strebenprofil 20x20x230mm
18	4	VI-0342		Strebenprofil 20x20x575mm
19	3		Schnappdeckelglas_hanger	

Erstellt durch:	Genehmigt von:	Gewicht:	
Carine Allen			

Montagebaugruppenbez.: Niryo Frame

FLUXANA®
XRF Application Solutions

Projekt:

Artikelnr.:

Montagebaugruppenzeichnungsnr.:

Format:	Maßstab:	Änd.:	Datum:	Spr.:	Blatt
A4		0	21/06/2023	de	5/5

Bibliography

[1] Automatic Addison. "The Ultimate Guide to Jacobian Matrices for Robotics". In: *Automatic Addison* (2020). URL: https://automaticaddison.com/the-ultimate-guide-to-jacobian-matrices-for-robotics/.

[2] Robert O Ambrose et al. "Robonaut: NASA's space humanoid". In: *IEEE Intelligent Systems and Their Applications* 15.4 (2000), pp. 57–63.

[3] Arsalan Anwar. *simple steps to create and build your first ROS package*. https://medium.com/swlh/7-simple-steps-to-create-and-build-our-first-ros-package-7e3080d36faa. Published in. 2021,

[4] Andreas Aristidou et al. "Inverse kinematics techniques in computer graphics: A survey". In: *Computer graphics forum*. Vol. 37. 6. Wiley Online Library. 2018, pp. 35–58.

[5] Pritha Bhandari. "Independent vs. Dependent Variables | Definition & Examples". In: *Scribbr* (2022). URL: https://www.scribbr.com/methodology/independent-and-dependent-variables/.

[6] Stephen Brawner. *SolidWorks to URDF Exporter*. http://wiki.ros.org/sw_urdf_exporter. 2021.

[7] Tessa Brazda. *NASA Robonaut first generation*. https://www.nasa.gov/technology/r1-the-first-generation/. found in. 2023.

[8] Peter Brouwer. "Theory of XRF". In: *Almelo, Netherlands: PANalytical BV* (2006).

[9] Praneel Chand. "Developing a Matlab Controller for Niryo Ned Robot". In: *2022 1st International Conference on Technology Innovation and Its Applications (ICTIIA)*. IEEE. 2022, pp. 1–5.

[10] Ned Chapin. "Flowchart". In: *Encyclopedia of computer science*. 2003, pp. 714–716.

[11] Davet Coleman. *rqt_graph*. https://wiki.ros.org/rqt_graph. (2018).

[12] David Coleman et al. "Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study". In: (Apr. 2014).

[13] Peter Corke. "Integrating ros and matlab [ros topics]". In: *IEEE Robotics & Automation Magazine* 22.2 (2015), pp. 18–20.

[14] John J Craig. *Introduction to robotics mechanics and control*. Harlow: Pearson, 2022. fourth edition, global edition., 2022. ISBN: 978-1-292-16493-9.

[15] Deeptanshu D and Shubham Dogra. "Research Hypothesis: Definition, Types, Examples and Quick Tips". In: *Typeset* (Sept. 2022). URL: https://typeset.io/resources/how-to-write-research-hypothesis-definition-types-examples-and-quick-tips/.

[16] Anish Devasia. *PLC Sequencer Logic: An Overview*. https://control.com/technical-articles/plc-sequencer-logic-an-overview/. Aug. 2020.

[17] Ahmed El-Sherbiny, Mostafa A Elhosseini, and Amira Y Haikal. "A comparative study of soft computing methods to solve inverse kinematics problem". In: *Ain Shams Engineering Journal* 9.4 (2018), pp. 2535–2548.

[18] Farbod Fahimi. *Autonomous robots*. Springer, 2009.

[19] Carol Fairchild and Dr. Thomas Harman. *ROS Robotics By Example*. Packt Publishing, June 2016. ISBN: 978-1-78217-519-3.

[20] Tully Foote. *Topics - ROS Wiki*. http://wiki.ros.org/Topics. 2019.

[21] Marc-Henri Frouin. *Ned2 Documentation*. https://docs.niryo.com/product/ned2/v1.0.0/en/source/introduction.html. found in. 2022.

[22] Marc-Henri Frouin. *Niryo Company*. https://niryo.com. found in. 2017.

[23] Marc-Henri Frouin. *package Documentation*. https://docs.niryo.com/dev/ros/v4.1.1/en/source/stack/high_level.html. found in. 2022.

[24] Patrick Goebel. *ROS By Example*. Ed. by Lulu. Lulu, 2013. URL: \url{http://www.lulu.com/shop/r-patrick-goebel/ros-by-example-indigo-volume-1/ebook/product-22015937.html}.

[25] Posted Keerthana Gopalakrishnan and Kanishka Rao. "RT-1: Robotics Transformer for real-world control at scale". In: *Robotics at Google* (Dec. 2022). URL: https://blog.research.google/2022/12/rt-1-robotics-transformer-for-real.html.

[26] Todd Helmenstine. "Difference Between Independent and Dependent Variables." In: *ThoughtCo* (2022). URL: https://www.thoughtco.com/independent-and-dependent-variables-differences-606115.

[27] Niall Henry. "Understanding Robot Coordinate Frames and Points". In: *SolisPLC* (2022). URL: https://www.solisplc.com/tutorials/robot-coordinate-frames-and-points#conclusion.

[28] Damith Herath, Adam Haskard, and Niranjan Shukla. "Software Building Blocks: From Python to Version Control". In: *Foundations of Robotics: A Multidisciplinary Approach with Python and ROS*. Springer, 2022, pp. 85–88.

[29] Damith Herath and David St-Onge. *Foundations of Robotics: A Multidisciplinary Approach with Python and ROS*. Springer, 2022.

[30] Intel. *Types of Robots: How Robotics Technologies Are Shaping Today's World*. https://www.intel.com/content/www/us/en/robotics/types-and-applications.html. found in. 2023.

[31] "International Standard ISO-17506:2022". In: *COLLADATM digital asset schema specification for 3D visualization of industrial data*. Industrial automation systems and integration. 2022.

[32] Reza N. Jazar. *Theory of Applied Robotics: Kinematics, Dynamics, and Control*. 3rd edition. Cham: Springer International Publishing, Imprint: Springer, 2022. DOI: 10.1007/978-3-030-93220-6.

[33] Lentin Joseph. *Learning Robotics using Python*. Ed. by Packt Publishing. Packt Publishing, May 2015. ISBN: 9781783287536. URL: \url{https://www.packtpub.com/application-development/learning-robotics-using-python}.

[34] Lentin Joseph. *ROS Robotics Projects*. Packt Publishing, 2017. ISBN: 978-1-78355-471-3.

[35] Lentin Joseph and Aleena Johny. "Getting Started with Ubuntu Linux for Robotics". In: *Robot Operating System (ROS) for Absolute Beginners: Robotics Programming Made Easy*. Springer, 2022, pp. 1–52.

[36] Lentin Joseph and Aleena Johny. "Programming with ROS". In: *Robot Operating System (ROS) for Absolute Beginners: Robotics Programming Made Easy*. Springer, 2022, pp. 173–240.

[37] Ltd. Kawasaki Heavy Industries. *Comparison between robot and human movement*. https://robotics.kawasaki.com/ja1/xyz/en/1804-03/. 2018.

[38] Charles C Kemp et al. "Humanoids". In: *experimental psychology* 56 (2009), pp. 1–3.

[39] Jay LaCroix. *Mastering Ubuntu Server: Master the art of deploying, configuring, managing, and troubleshooting Ubuntu Server 18.04*. Packt Publishing Ltd, 2018.

[40] B.A. Lee M. "Robotics Background Information and Facts". In: *Salem Press Encyclopedia of Science* (2022). URL: https://bvyg.short.gy/robotics-background.

[41] Kevin M Lynch and Frank C Park. *Modern robotics*. Cambridge University Press, 2017. URL: www.cambridge.org/9781107156302.

[42] Alexei Makarenko, Alex Brooks, and Tobias Kaupp. "On the benefits of making robotic software frameworks thin". In: *International Conference on Intelligent Robots and Systems-Workshop for Measures and Procedures for the Evaluation of Robot Architectures and Middleware at IROS'07*. Vol. 2. 2007.

[43] J Michael McCarthy. *Introduction to theoretical kinematics*. MIT press, 1990.

[44] Mark Moll et al. "Teaching motion planning concepts to undergraduate students". In: (2011). DOI: 10.1109/arso.2011.6301976.

[45] Habib Oladepo. *Nodes - ROS Wiki*. http://wiki.ros.org/Nodes. (2018).

[46] Janis Osis and Uldis Donins. "Chapter 1 - Unified Modeling Language: A Standard for Designing a Software". In: *Topological UML Modeling*. Ed. by Janis Osis and Uldis Donins. Computer Science Reviews and Trends. Boston: Elsevier, 2017, pp. 3–51. ISBN: 978-0-12-805476-5. DOI: https://doi.org/10.1016/B978-0-12-805476-5.00001-0. URL: https://www.sciencedirect.com/science/article/pii/B9780128054765000010.

[47] Jia Pan, Sachin Chitta, and Dinesh Manocha. "FCL: A general purpose library for collision and proximity queries". In: *2012 IEEE International Conference on Robotics and Automation*. IEEE. 2012, pp. 3859–3866.

[48] Richard P Paul. *Robot manipulators: mathematics, programming, and control: the computer control of robot manipulators*. Richard Paul, 1981.

[49] Martin Pecka. *Urdf: Universal robotic description format*. http://wiki.ros.org/urdf. 2023.

[50] Jan Perez Ruiz Alexander Rosell Gratacòs. "A roadmap to robot motion planning software development". In: *Computer applications in engineering education* 18.4 (2010), pp. 651–660. DOI: 10.1002/cae.20269.

[51] Michael Peshkin and J Edward Colgate. "Cobots". In: *Industrial Robot: An International Journal* 26.5 (1999), pp. 335–341.

[52] Donald Lee Pieper. *The kinematics of manipulators under computer control*. Stanford University, 1969.

[53] Morgan Quigley, Brian Gerkey, and William D Smart. *Programming Robots with ROS: a practical introduction to the Robot Operating System*. " O'Reilly Media, Inc.", 2015.

[54] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. Kobe, Japan. 2009, p. 5.

[55] Open Robotics. *ROS*. https://www.ros.org/. (2021).

[56] Open Robotics. *ROS-documantation*. https://docs.ros.org/. 2015.

[57] Universal Robots. "Explanation on Robot Orientation". In: *Collaborative Robotic Automation* (Oct. 2023). URL: https://www.universal-robots.com/articles/ur/application-installation/explanation-on-robot-orientation/.

[58] Isaac Saito. *Actionlib - ROS Wiki*. http://wiki.ros.org/actionlib. (2018).

[59] Rainer Schramm. *Fluxana Boramat 18 Machine*. https://fluxana.com/products/fusion/boramat-18. 2002.

[60] Rainer Schramm. *Fluxana Boramat Mono Machine*. https://fluxana.com/products/fusion/boramat. 2002.

[61] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

[62] Ruben Smits, H Bruyninckx, and E Aertbeliën. *KDL: Kinematics and dynamics library*. https://www.orocos.org/kdl. found in. 2011.

[63] Andrew Spielberg et al. "Functional co-optimization of articulated robots". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2017, pp. 5035–5042.

[64] Adam A. Stokes et al. "A Hybrid Combining Hard and Soft Robots". In: *Soft Robotics* 1.1 (2014), pp. 70–74. DOI: 10.1089/soro.2013.0002.

[65] Ioan A. Sucan and Sachin Chitta. *MoveIt*. https://moveit.ros.org/. found in. 2013.

[66] Niryo Team. *Control Ned with an Arduino Board*. https://docs.niryo.com/applications/ned/v1.0.3/en/source/tutorials/control_ned_arduino.html. found in. 2022.

[67] Niryo Team. *Ned2 Overview*. https://niryo.com/products-cobots/robot-ned-2/. found in. 2022.

[68] Niryo Team. *PyNiryo OpenCV Example*. https://docs.niryo.com/dev/pyniryo/v1.1.2/en/source/vision/image_processing_overview.html. found in. 2023.

[69] Niryo Team. *Visual picking with Artificial Intelligence using TensorFlow*. https://niryo.com/learn-robotics/visual-picking-with-ai-tensor-flow/. found in. 2023.

[70] Ricardo Téllez, Alberto Ezquerro, and Miguel Ángel Rodríguez. *ROS in 5 Days: Entirely Practical Robot Operating System Training*. Independently published, 2016. ISBN: 1520138733.

[71] Ricardo Tellez, Alberto Ezquerro, and Miguel Angel Rodriguez. *ROS Manipulation in 5 Days: Entirely Practical Robot Operating System Training*. Independently published, 2017. ISBN: 1549531964.

[72] Inc. Teradyne. *Universal Robots*. https://www.universal-robots.com/. found in. 2005.

[73] METTLER TOLEDO. *METTLER TOLEDO ME BALANCE*. https://bvyg.short.gy/me_balance. 2020.

[74] Frantisek Trebuňa. *Applied mechanics and mechatronics*. ISSN 1662-7482. Trans Tech Publications, 2014. ISBN: 3-03826-572-1.

[75] Günter Ullrich et al. "Automated guided vehicle systems". In: *Springer-Verlag Berlin Heidelberg. doi* 10 (2015), pp. 978–3.

[76] Lucas Walter. *roscore*. https://wiki.ros.org/roscore. (2019).

[77] Yanqing Wu. *ROS-Master*. https://wiki.ros.org/Master. (2018).

[78] Keenan A Wyrobek et al. "Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot". In: *2008 IEEE International Conference on Robotics and Automation*. IEEE. 2008, pp. 2165–2170.

[79] Tatu Ylonen. "SSH–secure login connections over the Internet". In: *Proceedings of the 6th USENIX Security Symposium*. Vol. 37. 1996, pp. 40–52.

[80] Dejun Zhang, Haochen Wang, and Chunmei Lin. "The application of fuzzy control in high accurate automatic powder dosing system". In: *2011 International Conference on Electric Information and Control Engineering*. IEEE. 2011, pp. 1771–1774.

[81] Hui Zhong and Zhongwen Ye. "The research and manufacture of the control system in automatic dosing system". In: *The mechanic and electronic engineering technology*. Vol. 36. 2007, pp. 71–72.